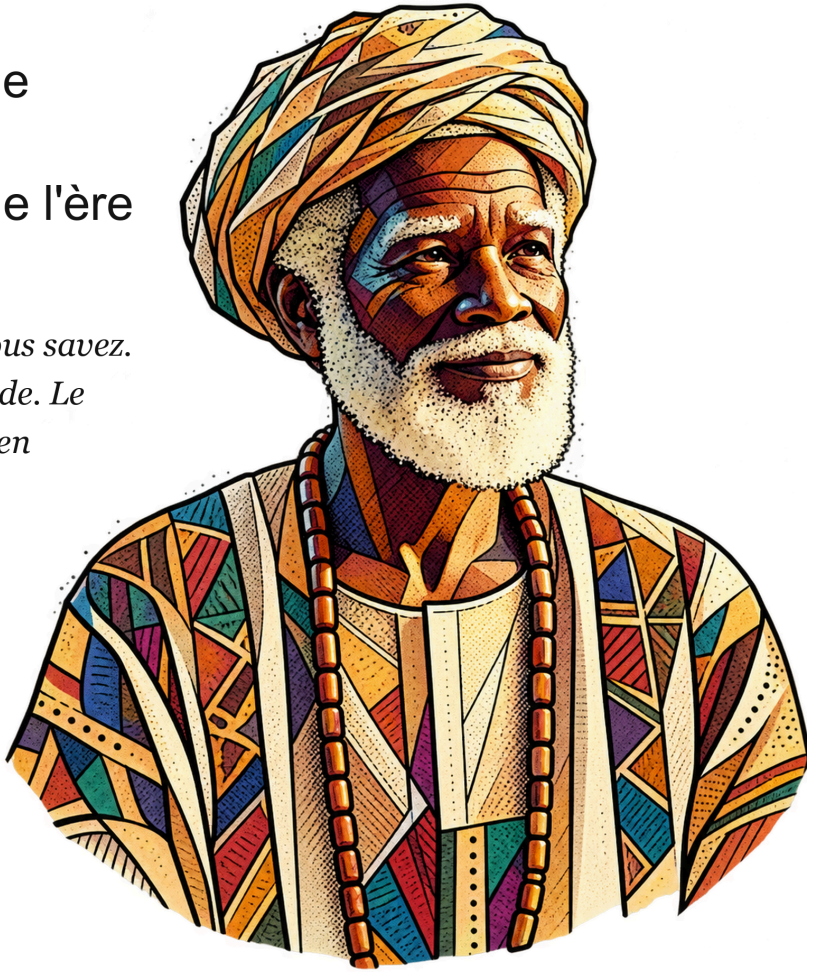
AYOUNI

Softanza

La plateforme logicielle gouvernée de l'ère de l'IA

Déclarez ce que vous savez. Un tribunal le garde. Le modèle d'IA vient en dernier.



MANSOUR AYOUNI

TUNIS · NIAMEY · PARIS · 2026

Softanza

Softanza

La plateforme logicielle gouvernée de l'ère de l'IA

Déclarez ce que vous savez. Un tribunal le garde. Le modèle d'IA vient en dernier.

Mansour Ayouni

AYOUNI

TUNIS · NIAMEY · PARIS · 2026

Épreuve de travail : 42 pages en brouillon, 0 en relecture, 0 publiées, sur 42. Cette édition française est une traduction des brouillons anglais.

Le code de chaque bloc a été exécuté dans la bibliothèque au commit 93d7a39bd : 285 blocs sur 285 ont tenu leur promesse. Les verdicts imprimés sont ceux de l'exécution, jamais réécrits.

Imprimé le 2026-10-17. Premier des livres AYOUNI.

© Mansour Ayouni · AYOUNI

Sommaire

Préface COLONNE 9

PARTIE I · L'ÂGE, L'ARTISAN, LA MÉTHODE

1 Le changement arrive plus vite que la confiance COLONNE 17
2 L'artisan du logiciel COLONNE 27
3 La méthode, depuis les premiers principes COLONNE 43
4 La plateforme en une image COLONNE 56

PARTIE II · LA PROGRAMMATION, REPENSÉE

5 Loi : le code adhère à la façon dont une personne pense, et le tribunal le juge quel que soit celui qui l'a écrit COLONNE 72
6 Le modèle mental, et les métaphores 86
7 Le style d'écriture, et les sept buts 100
8 Les fonctions comme expressions linguistiques, et la portée 120
9 Les motifs de pensée : le Softanzuter, Reaxis et les moteurs 136
10 La programmation devient raffinement 151

PARTIE III · LES LANGAGES, REPENSÉS

11 Loi : un langage est déclaré et fermé, et ce qui le parle, personne ou modèle, est jugé par lui COLONNE 166
12 Un langage, déclaré 181
13 Naturel, et exécutable : deux directions, une seule discipline 200
14 La programmation de la connaissance 222
15 Les langages de motifs 241
16 Sept langages, une porte 258

PARTIE IV · L'INTELLIGENCE, REPENSÉE

17 Loi : programmatique par défaut, le modèle en dernier, l'admission plutôt que la génération COLONNE 277
18 Les cinq registres, et les conversations qui demandent 291
19 Le modèle de langage de domaine 312
20 Des agents qui ne peuvent pas vous nuire 326
21 Le tribunal : promesses, perception, sécurité et performance 347
22 Comparé, honnêtement 366

PARTIE V · LA PLATEFORME, ASSEMBLÉE

23 Un monde, une constellation, une plateforme, et l'architecte 382
24 Répéter, puis valider 404

25 Vos utilisateurs gardent leur médium 428
26 L'environnement 449
27 Le moteur sous-jacent : trois couches, un moteur, Haro 470

PARTIE VI · DES ARTISANS DU LOGICIEL, FORGÉS

28 Un restaurant à Lyon 491
29 Une banque à Niamey 507
30 Une organisation sur le terrain 521
31 Ce que les projets montrent, et la suite 537

PARTIE VII · CE QUI VIENT

32 Beaucoup d'esprits sur un seul ouvrage COLONNE 559
33 La ligne suivante : une solution sur son propre moteur COLONNE 572

ANNEXES

A Le registre des innovations 585
B Les quatorze règles de la maison 592
C Les vingt-deux verbes et la constitution 597
D Le glossaire : les mots nomment des états, jamais des personnes 602
E Les gardes et les fixtures citées, par chapitre 609
F Les narrations dont le livre part 622
G Le livre comme données, pour l'agent qui le lit 631
H L'histoire du projet : de Ring à Haro 636

PRÉFACE · COLONNE

Préface

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 1 bloc(s) sur 1 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

UNE QUESTION QUE VOUS AVEZ DÉJÀ

Vous utilisez un outil qui écrit du code. Vous lui demandez une fonction, ou une fonctionnalité entière, et il répond en quelques secondes. Vient alors la question qu'aucun outil ne résout pour vous : ce code doit-il entrer dans mon système ?

Vous avez toujours eu des moyens de répondre. Vous lancez les tests. Le vérificateur de types est d'accord, ou il ne l'est pas. Un collègue relit le changement. Ces moyens ont été faits quand on écrivait le code lentement. Aujourd'hui une seule phrase peut produire mille lignes, et personne n'a le temps de relire mille lignes pour chaque phrase.

Le code est devenu bon marché. La confiance, non. Ce livre revient sans cesse à une phrase, celle sur laquelle repose chaque chapitre : **le changement arrive plus vite que la confiance.**

L'IDÉE, EN DEUX GESTES

D'abord vous **déclarez**. En texte simple, vous dites ce qu'est une chose : une application (je l'appelle un *monde*), un langage, un agent d'IA, un produit entier. Ensuite vous **gouvernez**. De votre déclaration vous obtenez un juge, que j'appelle un *tribunal*. Un tribunal est un validateur construit à partir de votre déclaration. Il rend un verdict, *tenu* ou *refusé*, et la raison, dans les mots de votre déclaration.

Chaque changement passe devant le tribunal avant d'entrer, quel qu'en soit l'auteur : vous, un collègue, un script ou un modèle d'IA. Le modèle peut toujours tout proposer. Il ne peut rien décider. On l'interroge aussi en dernier : tout ce que des règles peuvent décider, les règles le décident d'abord.

UNE DÉCLARATION, EN VRAI CODE

Voici le cas le plus petit. Vous ajoutez à votre système un agent d'IA, un programme qui surveille les niveaux de stock et réapprovisionne. Vous ne commencez pas par son code. Vous commencez par un court fichier texte qui dit ce qu'il est : son nom, la fréquence à laquelle il tourne, ce qu'il peut faire, et si ses actions peuvent être défaites. Ce fichier est une déclaration. Vous le chargez, et le tribunal le lit avant que l'agent puisse jamais tourner. Le fichier de cet agent omet quelque chose. Que dit le tribunal ?

HARO

```
cPia = "pia: 1
name: stock-watcher
kind: pi
reversibility: compensable
schedule:
  timer: 20
skills:
  - name: restock
    does: learn stock level ordered
"
oD = StzAgentDeclarationQ(cPia)
? oD.IsValid()
#--> 0
? oD.CiteFindings()
#--> [pia-coverage @ coverage] an agent must say WHAT IT COVERS. The engine loop refuses to
schedule an agent with no coverage statement (law 18), so a file without one could never run and
is refused here instead of at the first tick.
```

✓ exécuté : a affiché ce que ce bloc promet

Lisez-le en trois parties. Le texte entre les guillemets est la déclaration. La ligne suivante la charge. Ensuite `?` imprime une réponse, et la ligne après `#-->` est ce que le bloc a imprimé quand je l'ai lancé. Le tribunal répond `0` : refusé. Il nomme la règle et donne la raison : un agent doit dire ce qu'il couvre, parce que le moteur ne

planifiera pas un agent qui ne le dit pas, donc le fichier est refusé maintenant et non à sa première exécution. Quelques mots du fichier : `pia` nomme le format du fichier, `compensable` veut dire que ses effets peuvent être défaits par une action de compensation, et `law 18`, dans la raison du tribunal, est le numéro que le moteur donne lui-même à sa règle selon laquelle un agent sans couverture n'est jamais planifié.

Vous pourriez écrire ce contrôle dans votre propre CI. Ici personne n'a eu à le faire. Le tribunal des agents est livré avec la bibliothèque, donc la règle est celle de la bibliothèque, et les faits sont les vôtres, dans la déclaration. Un simple assert dans un script de démarrage juge un seul fait. Une déclaration tient dans un seul texte tous les faits sur l'agent, et le même tribunal la juge, qu'elle ait été écrite par une personne, un script ou un modèle d'IA. La règle peut aussi être la vôtre : le chapitre 2 a une règle que vous écrivez en quelques lignes sur deux faits, qu'un acteur est un modèle de langage et qu'il détient un effet, un pouvoir d'agir sur le monde comme envoyer un courrier, et le tribunal trouve la violation dans votre propre graphe d'agents. Cet exemple est petit exprès, et un schéma peut le vérifier aussi. Le livre ne soutient pas qu'un tribunal vérifie plus qu'un schéma. Il soutient qu'un seul tribunal, construit à partir de votre déclaration, se tient devant chaque acteur et donne sa raison. Il ne lit pas mille lignes générées pour les déclarer bonnes : il vérifie qu'un changement tient dans ce que vous avez déclaré, et le chapitre 24 montre un changement essayé sur une copie de votre système avant que rien ne soit validé. Pour une chose à vous, par exemple un langage que vous inventez, le chapitre 12 montre comment faire le tribunal, et dit jusqu'où cela est construit. Le chapitre 1 raconte une deuxième scène, avec un service de courrier, et le chapitre 2 relance ce bloc.

Vous direz que vous avez déjà des tests, des schémas et des contrôles de politique dans votre CI. C'est vrai. Aujourd'hui ces règles vivent à plusieurs endroits : des tests ici, un schéma là, un fichier de politique ailleurs, chacun écrit à la main après le code. Ici vous les énoncez une seule fois, dans la déclaration, et le tribunal en découle. Le chapitre 22 compare honnêtement les deux, et là où le terrain fait mieux, il le dit. Un tribunal juge ce que vous avez déclaré. Il ne peut pas juger ce que vous avez omis, et la limite de chaque chapitre dit où cela s'arrête.

CE QU'EST SOFTANZA

Une plateforme logicielle gouvernée. Vous déclarez ce que vous savez d'un monde, en texte clair ; un tribunal construit à partir de cette déclaration le garde ; et le modèle d'IA, quand il y en a un, vient en dernier. C'est la ligne de la couverture, et chaque chapitre la fait tourner. Matériellement, Softanza est trois choses, et pour chacune je dis jusqu'où elle existe.

- Une **bibliothèque** que vous appelez depuis du code. Vous écrivez quelques lignes de Haro comme celles ci-dessus, et vous les lancez avec la bibliothèque chargée. La bibliothèque tient les déclarations et les tribunaux qui en sont faits. Sa source est à github.com/mayouni/stzlib, et chaque bloc de code de ce livre a tourné sur un commit nommé de celle-ci. Si votre travail est en Python ou en TypeScript, vous l'utiliseriez aujourd'hui à côté de votre code, non dedans : par une seule porte gouvernée elle peut exécuter du code en six autres langages, Python, NodeJS (JavaScript), R, Julia, Prolog et C, qui avec l'hôte font les sept du chapitre 16.
- Un **moteur** sous la bibliothèque, écrit en Zig, un langage système. Il fait le calcul réel, une seule fois et en un seul endroit, pour que chaque langage qui lui parle obtienne le même comportement. Il tourne aujourd'hui, et les domaines de la bibliothèque y passent un à un : le chapitre 4 dit lesquels calculent encore dans le langage hôte.
- Un **langage**, Haro, fait pour tenir les déclarations. Pour l'instant il emploie la syntaxe de sa première facette, si bien que tout le code écrit jusqu'ici tourne sans changement. Son propre exécutable est en développement : il a une charte et un premier essai.

Le chapitre 4 dessine les trois en une seule image.

OÙ IL EN EST

Softanza n'est pas un projet de recherche, même s'il repose sur une recherche profonde. Des projets réels l'ont façonné : un restaurant, une banque et une organisation, et leurs retours ont façonné sa conception. Des parties de Softanza sont en production. D'autres sont en développement, comme il est normal pour une plateforme neuve et aussi ambitieuse. Chaque chapitre dit laquelle est laquelle.

CE QU'EST CE LIVRE

C'est un livre d'arguments, avec du code qui tourne. Pour chaque question sur la programmation, sur les langages et sur l'intelligence, je fais quatre choses. Je donne la réponse que donne d'ordinaire le terrain. Je donne la réponse que prend Softanza. Je donne la décision que j'ai prise, et ce que j'ai refusé. Et je donne la preuve : le code a tourné, et les tests qui le prouvent sont nommés. La troisième, la décision et ce que j'ai refusé, est la raison d'être de ce livre : le site montre ce qui est, et ce n'est qu'ici que vous trouverez ce qui a été refusé. Je dis aussi ce que le terrain fait déjà bien, pour que vous puissiez comparer.

Il compte sept parties et trente-trois chapitres. Vous n'avez pas besoin de connaître Softanza ni Zig. Il vous faut avoir écrit du code, et avoir ressenti le doute ci-dessus. Le site du projet tient ses preuves, avec leurs démonstrations, et son cours enseigne le même terrain par des exercices. Ce livre tient l'argument.

Chaque chapitre a six courtes portes, vers sa fin. Une porte est une entrée pour un genre de lecteur : le programmeur, l'analyste, le concepteur, le décideur, le formateur et l'agent. L'agent est un programme d'IA, parce que le livre est aussi publié sous forme de données qu'un programme peut lire. Entrez par la porte qui est la vôtre aujourd'hui.

COMMENT LE LIRE

Lisez d'abord neuf chapitres. Ils portent tout l'argument et peuvent se lire seuls : les quatre de la partie I, les deux derniers, et trois qui énoncent chacun une règle que le livre tient partout. Chapitre 5 : le code doit se lire comme une personne pense. Chapitre 11 : un langage est déclaré et fermé. Chapitre 17 : un programme est programmatique par défaut, ce qui veut dire que les règles et le code décident d'abord, et que le modèle d'IA vient en dernier. Le lecteur a un bouton qui n'affiche que ces neuf. Revenez aux autres quand l'un d'eux vous intéresse.

Chaque chapitre a les mêmes neuf parties, dans le même ordre. L'idée telle que le terrain la tient aujourd'hui. Le premier principe. La forme qu'elle prend dans Softanza, en code. La décision, et ce que j'ai refusé. Le tribunal : les tests nommés

qui prouvent l'affirmation du chapitre. Le terrain, et ce qu'il fait mieux. La limite. Les six portes. Un exercice, auquel vous répondez avant de le lancer. Les chapitres 5, 11 et 17 s'ouvrent par une courte section de plus, sur ce que l'ère de l'IA change.

Un nom de plus que vous rencontrerez. Softanza vous pose la question à laquelle vous devez répondre, dans vos propres mots, et refuse de deviner la réponse à votre place. J'appelle cette pratique le wise coding. Le chapitre 18 la fait tourner, registre par registre.

À QUOI SE FIER

Chaque bloc de code de ce livre a été lancé dans la bibliothèque. Sous chaque titre, la page dit quand, à quel commit, et combien de blocs ont tenu leur promesse : combien ont imprimé ce que dit la ligne après `#-->`. Si un bloc imprime autre chose, la construction échoue, et la page avec elle. Les exercices sont à vous de les lancer.

Deux mots reviennent sur bien des pages. Le *dossier* est l'ensemble des dépôts, des sessions et des fichiers sur lesquels ce livre a tourné, au commit imprimé sous chaque titre. L'*étude* est ma propre étude de ce dossier, faite avant la rédaction des chapitres. Quand une page dit que le dossier ne contient pas de réponse, elle parle de ces fichiers, à ce commit.

Quand un chapitre dit ce qui n'est pas fini, il emploie trois mots et aucun autre. *Construit* : le code existe, un test existe, et un bloc a tourné. *Conçu* : un document le dit, et rien ne tourne. *Absent* : j'ai cherché et je n'ai rien trouvé, et je dis jusqu'où j'ai cherché. Le site du projet emploie cinq noms pour les mêmes choses, et le chapitre 3 montre comment ils se ramènent à ces trois-là.

Une remarque de plus. L'exécutable propre de Haro est encore en développement, donc les blocs marqués `haro` tournent aujourd'hui sur le runtime du langage pour lequel la bibliothèque a d'abord été écrite, version 1.27, et chaque page le dit. L'annexe H, l'histoire du projet, raconte comment on en est arrivé là. Dans le code, les noms propres de la bibliothèque, ses classes, ses fichiers et ses clés, portent encore le nom de ce langage, et le porteront jusqu'à ce que la bibliothèque en soit retirée ; la prose, non.

À PROPOS DE CE BROUILLON

Les chapitres ont été rédigés avec des agents d'IA, à partir des propres fichiers du projet, et chaque bloc de code a été lancé. Chaque page porte son étape en haut : Brouillon, écrite, son code exécuté et jugé, lue par des lecteurs neufs ; En relecture, pendant que je la relis ; Publiée, une fois que je l'approuve. Aujourd'hui, chaque page est un brouillon. Là où les fichiers ne contenaient aucune réponse à une question, la page le dit en une phrase simple et n'en invente pas. L'édition française est une traduction des brouillons anglais, et chaque page française le dit. Cette préface a été écrite après les chapitres, et elle reste inachevée jusqu'à ce que j'écrive la mienne. C'est le premier livre de ma maison d'édition, AYOUNI. Le deuxième, *Refinement-Oriented Programming*, parle d'une discipline pour le logiciel à l'ère des agents d'IA, racontée à travers une banque.

PARTIE I

L'âge, l'artisan, la méthode

1 · Le changement arrive plus vite que la confiance
2 · L'artisan du logiciel
3 · La méthode, depuis les premiers principes
4 · La plateforme en une image

PARTIE I · CHAPITRE 1 · COLONNE

Le changement arrive plus vite que la confiance

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 2 bloc(s) sur 2 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Demandez à l'industrie ce qui a changé dans la programmation ces dernières années et vous obtiendrez une seule réponse, dite avec enthousiasme ou avec effroi : *les machines écrivent le code maintenant*.

L'image est familière. Un modèle d'IA propose une fonction, un fichier, une application entière. Une personne la relit, ou non. On ajoute des outils autour du modèle pour le rendre plus sûr : des linters qui lisent ce qu'il a écrit, des tests qu'il génère lui-même, des garde-fous qui attrapent le pire, un second modèle qui juge le premier. Le modèle est au centre. Tout le reste est une clôture bâtie autour de lui après coup.

Je ne crois pas que cette image se trompe sur les faits. Elle se trompe sur l'endroit où se tenir.

2. LE PREMIER PRINCIPE

Voici la phrase à laquelle je reviens sans cesse, et c'est la plus ancienne phrase de ce livre : **le changement arrive plus vite que la confiance.**

Le code est devenu bon marché. Mille lignes coûtent une phrase. Ce qui n'est pas devenu bon marché, ce qui est même devenu la chose la plus rare du logiciel, c'est la confiance que ces lignes font ce que leur auteur, humain ou machine, dit qu'elles

font. Toutes les équipes que j'ai rencontrées ces deux dernières années ont plus de code qu'elles ne peuvent en relire et moins de confiance qu'avant. Le goulot de notre époque n'est pas l'écriture. C'est l'admission : décider ce qui peut entrer dans un système et y rester.

Aussi la question que pose Softanza n'est pas *comment générer mieux ?* C'est *comment admettre ?* Qui juge un changement, par quelle loi, avant qu'il s'exécute, et ce qu'on écrit du verdict pour que quelqu'un puisse le lire plus tard et en répondre.

Lorsque vous placez l'admission au centre, l'image se retourne. Le tribunal est au milieu. Le modèle est un auteur de propositions parmi d'autres, et il vient en dernier, après la personne, après les données, après la formule, après la phrase simple. Rien de ce qu'il dit n'entre dans le monde avant que le tribunal l'ait lu. Ce n'est pas une clôture bâtie autour d'un générateur. C'est un autre bâtiment.

3. LA FORME SOFTANZA

Laissez-moi vous montrer plutôt que dire, par l'exemple, ce qui a toujours été ma façon préférée d'enseigner.

Une solution déclare qu'elle dépend d'un service de courrier. Pendant que je programme, je lie cette dépendance à un bac à sable : un objet qui capture chaque message et n'envoie rien, pour que mes tests puissent lire ce qui *serait* parti. Gratuit, hors ligne, déterministe. Puis je déplace la solution en production, le bac à sable toujours lié, comme on le fait à deux heures du matin avant une démonstration.

HARO

```
oReg = new stzServiceRegistry("my-solution")
oReg.Bind(:mail, new stzMailSandbox())
oReg.SetPhase(:emulated)
? oReg.IsSound()
#--> 1
oReg.SetPhase(:production)
? oReg.IsSound()
#--> 0
? oReg.Findings()[1][:message]
#--> still bound to a SANDBOX in a production phase -- a fake must never ship
```

✓ exécuté : a affiché ce que ce bloc promet

Le registre ne la déclare pas saine. En phase émulée il répond un ; en production il répond zéro, et la ligne suivante dit pourquoi : un faux ne doit jamais être livré. Le faux est refusé par son nom, et un service déclaré et jamais lié lève une erreur au moment où on le demande, au lieu de ne rien faire en silence. Je n'ai pas eu à me souvenir de le vérifier. La phase a décidé de ce qui revient, et la vérification est imposée, non mémorisée.

C'est toute la philosophie en une petite scène. L'expression est libre : liez ce que vous voulez, aussi longtemps que vous voulez. L'admission est gouvernée : la barrière lit la phase et la liaison, et refuse.

Maintenant une deuxième scène, parce que la première n'a pas de modèle d'IA dedans et que vous attendez le modèle.

Prenons un restaurant pour illustration : un agent d'IA qui connaît son monde propose de passer une commande. Dans Softanza, l'agent ne la passe pas. Il propose un plan. Un atelier répète le plan sur un jumeau du système, une copie qui ne garde aucune référence vers le réel, et enregistre chaque opération qu'il effectuerait. Un tribunal lit la répétition à la lumière des règles que le restaurant a déclarées : qui peut commander, ce qui peut être commandé, ce qu'un fournisseur doit confirmer. Alors seulement un acteur gouverné valide, et la validation est enregistrée avec son acteur et son issue. La garde qui prouve cette séquence, `safeworld_narrated`, répète dans un jumeau des fichiers, planifie et laisse un acteur valider ; elle compte soixante-seize assertions au moment où j'écris, dont aucune ne parle d'un restaurant, et le site montre une exécution enregistrée du même mécanisme.

Remarquez ce que le modèle a fait et n'a pas fait. Il a proposé. Il n'a pas décidé, pas écrit, pas envoyé. Il était un auteur de propositions à côté des registres de réponse. Trois d'entre eux, une option choisie, une structure de données donnée, une phrase dite, sont construits aujourd'hui et ouverts à la personne avant même que le modèle soit appelé ; deux autres, une formule écrite et un exemple montré, ne sont pas encore construits (chapitre 18). J'appelle cela *programmatique par défaut, le modèle en dernier*. C'est la phrase sous chaque chapitre de la partie IV.

Et une troisième scène, la plus petite, parce qu'elle montre la même loi au niveau d'une seule ligne. La plateforme est jugée sur l'allure de son code et sur ce qu'il répond :

HARO

```
o1 = new stzList([ "tea", "rice", "tea", "fish", "rice", "tea" ])
? @@( o1.FindAll("tea") )
#--> [ 1, 3, 6 ]
? @@( o1.DuplicatesRemoved() )
#--> [ "tea", "rice", "fish" ]
```

✓ exécuté : a affiché ce que ce bloc promet

La question est dite comme elle est pensée : trouver tout, retirer les doublons. Et les deux lignes sous les deux questions ne sont pas mon commentaire. Ce sont des promesses que la bibliothèque a tenues la nuit où ce chapitre a été construit. Si elle cesse de les tenir, cette page échoue. La même loi, l'admission avant la confiance, appliquée à un paragraphe.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Chaque chapitre de ce livre porte cette section, et j'ai promis dans la préface que c'est la raison d'être du livre. Soyons donc concret sur les décisions derrière ce premier chapitre, car elles n'ont pas été confortables.

J'ai refusé de mettre le modèle d'IA au centre. Quand les modèles d'IA sont arrivés, c'était le geste évident, et j'avais les pièces pour le faire : une couche naturelle, un graphe de connaissances, une classe d'agent. J'aurais pu brancher un modèle dans chacune et livrer un Softanza qui « parle ». Je ne l'ai pas fait, et j'ai écrit ce refus dans l'architecture comme une loi : l'esprit est programmatique par défaut, et un modèle ne peut que proposer. J'ai renoncé à un peu de mode par ce choix. Je crois avoir gardé la plateforme.

J'ai mis à la retraite le langage sur lequel je suis né. Softanza a été écrit pendant des années sur un petit langage de script chaleureux que j'aime et que j'ai supplié, bien avant l'ère des LLM, de faire du langage naturel exécutable sa première préoccupation. Il ne l'a pas fait, et c'est son droit. En 2026, j'ai décidé que le langage ne serait pas porté plus loin : le runtime sous Softanza est en train d'être remplacé par un runtime qui nous appartient, écrit en Zig, une charte et un spike pour l'instant, portant l'ancienne surface pour que chaque ligne que j'ai écrite puisse continuer à s'exécuter, sous un nouveau nom, **Haro**. L'ancienne surface reste parce que le code est le patrimoine ; l'ancien runtime s'en va parce que la souveraineté sur le runtime est l'un des engagements que j'ai pris. Cette décision n'a pas été gratuite. Elle est au chapitre 27 avec ses chiffres, y compris celui que je ne voulais pas : notre runtime est plus lent que celui contre lequel nous l'avons mesuré, d'un facteur entre un et deux, sur chaque ligne. J'ai consigné la perte dans le mémo du domaine de projets du 13 septembre. Un livre qui cache cela ne mérite pas votre confiance, et une plateforme non plus.

J'ai refusé de laisser quoi que ce soit entrer dans BASE sans un tribunal, et j'ai échoué. Les trois couches de la bibliothèque, NOYAU, BASE et MAX, devaient garder le fondement petit et les innovations au-dessus. Sous la pression du

temps, j'ai laissé tout aller dans BASE : quatre-vingt-treize pour cent du code s'y trouve aujourd'hui, et la règle de dépendance est tenue au niveau du moteur et rompue au niveau de l'héritage. Je le dis au premier chapitre, non au dernier, parce que la discipline que je m'apprête à prêcher pendant trente-deux chapitres est une discipline que j'ai enfreinte, et l'étude du dossier m'a pris sur le fait. C'est à cela que sert un tribunal, et celui des couches n'est encore que proposé.

J'ai refusé une seconde intégration. Quand quelqu'un demande « et Python, et JavaScript, et mon langage », la réponse est une porte, à gouverner, et jamais un autre runtime hôte cousu dans la plateforme. Les sept langages du dossier, l'hôte et six runtimes derrière une seule porte, ont été choisis pour sept couches d'une solution, et le chapitre 16 les compte. La porte est la réponse à « et le langage X », et c'est une seule réponse, pas vingt.

Ce qui a été refusé fait autant partie de Softanza que ce qui a été construit. Vous ne trouverez pas les refus sur le site, parce que le site montre ce qui est. Vous les trouverez ici, dans chaque chapitre, sous ce titre.

5. LE TRIBUNAL

Qu'est-ce qui prouve ce chapitre ? Trois choses, et je vous les donne par leur nom pour que vous puissiez les vérifier plutôt que me croire.

Le refus d'un faux en production par le registre de services est une garde, `service_registry_narrated`, dans le volet système. La suite proposer, répéter, juger et valider de l'agent d'IA est `safeworld_narrated`, soixante-seize assertions, dans le volet agentique, à côté de `governance_crossing_narrated`, dans le volet système, pour la traversée elle-même. Les blocs de code de cette page ont été exécutés par l'exécuteur propre du livre à l'intérieur de la bibliothèque la nuit où cette page a été construite, et sa sortie est montrée telle qu'elle est venue.

Trois gardes pour un chapitre de philosophie peuvent sembler peu. Ce sont les trois bonnes, parce que ce sont les trois endroits où la philosophie rencontre un verdict.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`service_registry_narrated` 93 assertions
base/test/system/service_registry_narrated.ring

`safeworld_narrated` 76 assertions
base/test/agentic/safeworld_narrated.ring

`governance_crossing_narrated` 31 assertions
base/test/system/governance_crossing_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Je ne ferai pas semblant que le terrain n'a rien. Il a, ces trois dernières années, construit les générateurs de code les plus capables de l'histoire, et je les utilise chaque jour. La réponse de l'industrie au problème de la confiance est un vrai travail : tests générés, robots de revue, moteurs de politiques, bacs à sable pour agents d'IA, sorties structurées, un second modèle pour noter le premier. Là où le terrain mène, et il mène nettement, c'est dans les générateurs eux-mêmes. Softanza ne construit pas les grands générateurs et ne les concurrence pas ; là où il entraîne, il entraîne de petits modèles liés à un seul domaine, dont chaque réponse sera vérifiée par une grammaire. Le terrain mène aussi en échelle : ses outils de revue lisent des volumes de code généré contre lesquels rien dans Softanza n'a été exécuté.

Là où je crois que Softanza se tient seul, c'est dans l'ordre des choses. Le terrain ajoute l'admission à un générateur. Softanza ajoute un générateur à un tribunal. Dans le terrain, un modèle d'IA écrit dans votre langage et des outils le vérifient ensuite ; ici, le modèle ne peut écrire que dans un langage que vous avez déclaré, dont la grammaire est close et dont le tribunal est destiné à naître avec lui, de sorte que, pour les langages dont la grammaire est compilée aujourd'hui, une phrase

invalide ne peut pas du tout être prononcée. Dans le terrain, le dialogue avec un système est un prompt ; ici, le système demande, et la réponse peut être un exemple, une formule ou une phrase avant d'être jamais un modèle. Dans le terrain, la trace d'une décision est un journal ; ici, c'est une transcription avec un locuteur et une certitude, dont on répond pour toujours.

Ce sont des affirmations, et chacune a son chapitre et sa garde. Je les liste ici pour que vous sachiez où va l'argument, et pour que vous puissiez me le faire tenir.

7. LA LIMITE

CONSTRUIT le tribunal qui admet ou refuse un changement, pour une personne et pour une machine

ABSENT une plateforme cliente qui tourne sur le moteur Softanza

La limite honnête de ce chapitre est aussi la limite honnête de toute la plateforme, et je la mets sur la première page à dessein.

Softanza a été utilisé. Une banque à Niamey fait tourner une plateforme d'évaluation dont la construction a façonné Softanza, sous licence, attestée par le guide d'installation que nous avons livré. Un restaurant à Lyon fait tourner une plateforme pour le commerce de quartier dont le propriétaire a fait seul la première démonstration, sur son propre téléphone, avec une application de messagerie comme ligne d'assistance. Une organisation au Niger a son hub conçu. Leurs leçons sont dans le code, par nom de fichier, dans la partie VI.

Mais les plateformes clientes tournent aujourd'hui sur des piles ordinaires. Celle de la banque est une application Python ; celle du restaurant est un ensemble de mondes web d'un seul fichier ; le hub de l'organisation n'est pas construit. Leurs déclarations sont en cours d'écriture comme celles de Softanza elle-même, de sorte que les faire passer sur le moteur Softanza soit un changement de moteur et non de plan. Ce passage est l'étape suivante. Le tribunal est construit, et les plateformes existent ; le prochain jalon est la première d'entre elles tournant sur le moteur. C'est la ligne suivante que ce livre portera, et le chapitre 33 ne parle de rien d'autre.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous garderez votre éditeur et votre langage. Ce qui change, c'est que chaque affirmation que vous faites sur votre code peut porter une promesse dessous, et un tribunal la lira.
L'analyste	Vos règles deviennent des données qu'une personne peut changer, et chaque changement est jugé avant d'atterrir et enregistré après.
Le concepteur	L'interface est jugée aussi : vingt-deux verbes, cent vingt-deux règles, et un vérificateur qui refuse ce qu'une machine peut vérifier.
Le décideur	La question que vous posez à chaque fournisseur, « qui décide, et qu'est-ce qui est écrit », a ici une réponse mécanique, et c'est la même réponse pour une personne et pour une machine.
Le formateur	Le tuteur demande et ne répond jamais le premier. C'est une loi de la plateforme, non une fonction d'un produit.
L'agent	Vous êtes le bienvenu ici, et vous serez jugé. L'annexe G est la vôtre.

9. EXERCICE

Prenez une fonction qu'un modèle d'IA a écrite pour vous cette semaine. Dessous, écrivez la ligne qu'elle doit imprimer pour une entrée qui vous importe, sous la forme que vous venez de rencontrer :

HARO

```
? YourFunction(the_input)
#--> the line it must print
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-la. Si la ligne correspond, vous avez votre première promesse. Sinon, vous avez votre premier constat, et vous avez appris sur cette fonction en une minute plus que l'explication du modèle ne vous en a dit. Ne la corrigez pas encore. Le chapitre 2 parle de qui vous êtes quand vous le faites.

OÙ SONT LES PREUVES

SUR LE SITE	forged
SUR LE SITE	agentic
DÉCISIONS DE L'ÉTUDE	12.10, 12.11, 12.15, 12.28, 12.29

PARTIE I · CHAPITRE 2 · COLONNE

L'artisan du logiciel

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends le mot « crafter » au terrain, à son artisanat et à ses deux réponses qui définissent la personne par un outil, le low-code et l'agent de code. Je dis ce que le site entend par lui. Je montre les six portes, chacune avec une ou deux lignes qui s'exécutent, et les trois cas pour lesquels ce livre est écrit : un restaurateur à Lyon, Softibank, une banque à Niamey, une organisation sur le terrain. Je dis ce que j'ai gardé, le programmeur en premier ; ce que j'ai ajouté, l'agent d'IA, avec un pouvoir retenu ; et ce que j'ai refusé, « maker » et « builder ». Le principe est une phrase qui joint celle du site et celle du glossaire : un artisan du logiciel est quiconque déclare ce qu'il sait d'un monde, et possède ce qu'il déclare.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a déjà un mot pour les programmeurs qui prennent soin de leur travail : l'artisanat (craftsmanship). Un livre de 2001 a donné son nom à l'idée, et un manifeste de 2009 lui a donné son slogan. Depuis, « software craftsman » désigne un professionnel qui tient son code à un niveau plus élevé que celui que le travail exige. Cela relève de la connaissance générale, non d'une lecture du dossier. C'est un bon mouvement. C'est aussi un mouvement de programmeurs, sur les standards des programmeurs.

Les deux autres réponses du terrain à « qui fait le logiciel, si ce ne sont pas les programmeurs » définissent la personne par un outil. La première est le low-code : des gens du métier qui construisent des applications sur la plateforme visuelle d'un

éditeur, le « développeur citoyen » des analystes. La seconde est l'agent de code : un modèle d'IA qui lit des fichiers, les modifie et lance des commandes, piloté en mots simples. Le « vibe coding » est le nom de la pratique qui consiste à garder tout ce qu'il écrit. Les deux réponses ont la même forme. La personne est définie par un outil, et le travail vit là où vit l'outil. Quand l'éditeur change ses prix ou son modèle d'IA, le travail de la personne change avec. Cela aussi relève de la connaissance générale, non d'une lecture du dossier.

La section suivante dit ce que le site entend par ce mot.

2. LE PREMIER PRINCIPE

Le site donne la définition en deux phrases. Depuis le 8 octobre 2026, elles disent « crafter » ; avant, elles disaient « maker ». Le glossaire dit qu'un artisan du logiciel est quiconque déclare, au moment où il déclare. Le site dit qu'il possède ce qu'il fait. Je garde les deux, en une phrase : **un artisan du logiciel est quiconque déclare ce qu'il sait d'un monde, et possède ce qu'il déclare : l'artefact est à lui, en texte brut, et il n'expire pas.** La compétence n'entre pas dans la définition. Le mot « artisanat » la suggère, et je garde le mot : la personne est définie par l'acte et par la possession, non par sa compétence de programmation. Je tiens cela pour une vue, non pour un résultat.

3. LA FORME SOFTANZA

J'ai été le premier artisan du logiciel de Softanza, et pas pour une raison que j'aurais choisie. Ma femme dirige un club de musique où l'on apprend à chanter des chansons classiques arabes. Chaque dimanche matin, elle m'envoie la liste des chansons à préparer, et je cherche leurs paroles. Les paroles trouvées en ligne répètent chaque phrase comme le chanteur l'a interprétée, et elle me rappelle, à chaque fois, d'en supprimer les doublons. J'ai écrit `RemoveDuplicates()` pour cesser de le faire à la main. Sa famille est-elle toujours dans la bibliothèque, et que dit-elle d'une liste qui contient trois fois « tea » ?

HARO

```
o1 = new stzList([ "tea", "rice", "tea", "fish", "rice", "tea" ])
? o1.ContainsDuplicates()
```

```
#--> 1
? o1.NumberOfOccurrence("tea")
#--> 3
```

✓ exécuté : a affiché ce que ce bloc promet

La famille répond : la liste contient des doublons (`1`), et `tea` apparaît `3` fois.

J'en parle dans un article sur ce sur quoi la bibliothèque a été construite : j'ai introduit ses fonctions quand un vrai problème les demandait, ou quand j'anticipais qu'elles répondraient aux besoins des programmeurs. Celle-ci est née d'un problème : l'artisan du logiciel vient d'abord, et la fonction après.

Depuis 2022, je présente Softanza non par son anatomie, ses modules et ses classes, mais par qui vous êtes et ce que vous essayez de faire. La présentation de 2022 avait cinq publics, chacun avec une recette de trois à cinq étapes. Le 9 octobre 2026, le site avait sept portes, une par sorte d'artisan du logiciel. La septième, pour une équipe qui livre une application, y a été ajoutée ce jour-là, et ce chapitre garde les six avec lesquelles il a été rédigé. Chaque porte dit ce que vous déclarez, ce que vous obtenez, et par où commencer. Voici ce que chacune des six déclare, en une ou deux lignes qui s'exécutent.

Le programmeur déclare l'intention dans une bibliothèque qui se lit comme une phrase. Vous venez de la voir.

L'analyste déclare les données d'un monde, et le moteur calcule la vue. Ici, trois régions du Niger avec la population que le recensement de 2012 a comptée :

HARO

```
o = new stzTable([ [ :region, :population ],
    [ "Agadez", 487620 ], [ "Maradi", 3402094 ], [ "Zinder", 3539764 ] ])
o.Show()
#--> ┌──────────┬────────────┐
#    │ Region   │ Population │
#    ├──────────┼────────────┤
#    │ Agadez   │     487620 │
#    │ Maradi   │    3402094 │
#    │ Zinder   │    3539764 │
#    └──────────┴────────────┘
```

✓ exécuté : a affiché ce que ce bloc promet

Le moteur a dessiné la vue à partir des seules données : `Maradi │ 3402094` en est une ligne.

Le concepteur déclare une image comme un programme, et obtient la même image à chaque fois :

HARO

```
? Q("Softanza").BoxedXT([ :AllCorners = :Round ])
#--> ╭──────────╮
#    │ Softanza │
#    ╰──────────╯
```

✓ exécuté : a affiché ce que ce bloc promet

L'image est la sortie du programme : un `Softanza` encadré, aux coins arrondis.

Le décideur déclare ce que chaque acteur du monde peut faire, une personne ou un agent d'IA, et le tribunal l'impose. Un agent est un petit fichier texte. Que dit le tribunal à celui qui ne dit pas ce qu'il couvre ? Chargez-le :

HARO

```
cPia = "pia: 1
name: stock-watcher
kind: pi
reversibility: compensable
schedule:
  timer: 20
skills:
  - name: restock
    does: learn stock level ordered
"
oD = StzAgentDeclarationQ(cPia)
? oD.IsValid()
#--> 0
? oD.CiteFindings()
#--> [pia-coverage @ coverage] an agent must say WHAT IT COVERS. The engine loop refuses to
schedule an agent with no coverage statement (law 18), so a file without one could never run and
is refused here instead of at the first tick.
```

✓ exécuté : a affiché ce que ce bloc promet

Le tribunal l'a refusé ! Sa raison est imprimée : `[pia-coverage @ coverage] an agent must say WHAT IT COVERS`. Ici « law 18 » est la règle du moteur selon laquelle un agent sans énoncé de couverture n'est jamais planifié, et `pia` nomme le format du fichier, comme le montre sa première ligne.

Le formateur déclare dans la propre langue de l'apprenant. Une phrase en haoussa s'exécute : elle demande un texte contenant `'hello'`, passé en capitales. Le moteur compte-t-il les lettres d'un mot arabe plutôt que ses octets, et connaît-il son écriture ?

HARO

```
? NaturallyIn("ha", "Yi rubutu dauke 'hello' Maida shi").Result()
#--> HELLO
? Q("سلام").NumberOfChars()
#--> 4
? Q("مرحبا بالعالم").Script()
#--> arabic
```

✓ exécuté : a affiché ce que ce bloc promet

La phrase en haoussa est revenue en `HELLO`. Le mot arabe compte `4`, ses lettres et non ses octets, et son écriture est `arabic`.

L'agent est le sixième artisan du logiciel, celui que la présentation de 2020 de la plateforme ne pouvait pas dessiner, selon les mots de la vision que j'ai ratifiée. Un agent d'IA se déclare dans le même genre de fichier, et il est le bienvenu, avec une seule chose retenue. Un agent dont l'esprit est un modèle de langage peut proposer ; il ne peut pas détenir d'effet. Qu'arrive-t-il à un fichier qui lui en donne un ?

HARO

```
cLlm = "pia: 1
name: rogue
kind: llm
coverage: proposes a summary
reversibility: reversible
schedule:
```

```
  timer: 20
proposes:
  prompt: Summarize {input}
  structure:
    - field: topic
      type: string
skills:
  - name: send
    when: always
    does: propose
    effect: send-email
"
oBad = StzAgentDeclarationQ(cLlm)
? oBad.IsValid()
#--> 0
? oBad.CiteFindings()
#--> [no-llm-effectful @ skills.send.effect] REFUSED: granting 'effectful' to llm actor 'rogue' -
- an LLM proposes, only a pi-gate commits (no-llm-effectful, enforced at CONSTRUCTION, not merely
audited).
```

✓ exécuté : a affiché ce que ce bloc promet

L'acteur llm `rogue` n'obtient pas son effet ! Le tribunal répond `REFUSED`, et dit quand : `enforced at CONSTRUCTION, not merely audited`. Un « pi-gate » est la barrière où un acteur gouverné valide.

Voici maintenant les trois cas pour lesquels ce livre est écrit. Ce ne sont pas des personas. Ce sont des gens et des organisations avec des documents, et la partie VI raconte chacun en entier. Ici, une page chacun.

Un restaurateur à Lyon. Il possède un restaurant de couscous, et le projet est une plateforme pour le commerce de quartier. Les notes du projet le disent sans détour : il n'est pas programmeur. Le projet a commencé par une offre en mai 2024 puis deux années de spécification riche qui n'ont rien produit qu'il puisse utiliser. Ce qui est sorti de cet échec est la règle selon laquelle je travaille depuis. Livrer petit, vite et utile, et laisser les fondations mûrir en parallèle sans jamais bloquer une livraison.

Le 4 juillet 2026, il a répondu à un rapport de cadrage dans ses propres mots et a renversé l'ordre du travail : client d'abord, restaurateur ensuite, fournisseur en dernier. L'architecture du projet a été redessinée en constellation de mondes, à

partir de la conception de la bibliothèque, et le premier avenant a été clos le 4 août 2026.

Une constellation est un graphe de mondes. Un monde est une application ; un lien est une relation déclarée entre deux mondes, pour un acte nommé, et rien ne passe sans lui. Le chapitre 23 montre les trois à l'exécution ; ici, ce n'est que la forme que le projet a prise.

Le projet compte onze itérations et une campagne de vingt-deux tests simulés derrière ses règles d'interface. Dans son propre rapport, la vitesse de commande est sa première priorité, non négociable. Et le 19 août, après une journée de tests sur le terrain avec deux téléphones et une application de messagerie, les notes l'appellent le co-architecte du projet.

La première démonstration, le 15 août 2026, a été la sienne seule : la liste de vérification du dossier la prévoit pour le propriétaire seul, avec son propre téléphone et une application de messagerie comme ligne d'assistance. Les mesures qu'il a envoyées par une application de messagerie sont celles du 19 août.

L'un de ses besoins est porté par une loi de la bibliothèque : essayer les paiements avant de payer quiconque pour un compte. Que dit le registre d'un faux service de paiement pendant que vous répétez, et que dit-il quand vous passez au réel ? Exécutez-le. Le registre est nommé une fois, et chaque ligne entre les accolades lui parle : le chapitre 5 appelle cette forme la scène.

HARO

```
oReg = new stzServiceRegistry("restaurant")
oReg {
        Bind(:payments, new stzPaymentsSandbox())
        SetPhase(:emulated)
        ? IsSound()
        #--> 1
        SetPhase(:production)
        ? IsSound()
        #--> 0
        ? Findings()[1][:message]
        #--> still bound to a SANDBOX in a production phase -- a fake must never ship
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `1` en phase émulée et `0` en production, et dit pourquoi : `still bound to a SANDBOX in a production phase -- a fake must never ship`. Un faux en production est refusé !

Une banque à Niamey. Une banque commerciale soumise aux règles de la banque centrale de l'union monétaire ouest-africaine est titulaire d'une licence pour un produit d'évaluation d'organisation que j'ai construit à côté de la bibliothèque. Il est installé sur son propre réseau interne, sous cette licence. Le guide d'installation livré à la banque est le document qui l'atteste. La leçon qu'il a laissée dans la bibliothèque est qu'une organisation peut être jugée par son propre organigramme, selon les règles d'un régulateur. Déclarez un organigramme avec un directeur général et un audit interne qui lui rend compte. Que disent les règles du régulateur à son sujet ?

HARO

```
oOrg = new stzOrgChart("bank")
oOrg {
        AddPositionXTT("ceo", "Chief Executive", [ :department = "executive" ])
        AddPositionXTT("audit", "Internal Audit", [ :department = "audit", :reportsTo = "ceo" ])
        ? @@( ValidateBCEAOGovernance() )
        #--> [ [ "status", "fail" ], [ "domain", "BCEAO_governance" ], [ "issuecount", 3 ], [
"issues", [ "BCEAO-001: No Board of Directors found", "BCEAO-002: Audit reports to non-board
position", "BCEAO-003: No dedicated Risk Management function" ] ] ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'organigramme échoue, avec `"issuecount", 3` : trois constats pour un organigramme de deux postes ! Chacun est une règle avec un numéro qu'un auditeur peut lire : `BCEAO-001: No Board of Directors found` est le premier. Le dossier montre la banque comme titulaire d'une licence d'un produit que j'ai construit ; il ne dit pas qui, à la banque, a déclaré quoi que ce soit.

Une organisation sur le terrain. Une organisation qui travaille au Niger, avec un siège à Niamey et des bases sur le terrain, a demandé un hub. Son propre cahier des charges préférait une plateforme low-code construite sur la suite bureautique d'un grand éditeur. Mon offre du 5 août 2026 a été choisie à la place. L'étude de

conception liste quatre gains, la possession en tête. L'organisation possède son vocabulaire et ses règles, les change sans programmer, ne paie pas de licence par poste, et peut tout remettre à un programmeur de son propre pays. L'étude de conception a été construite avec ses équipes entre le 22 septembre et le 8 octobre 2026. Deux de ses principes sont ceux du livre en petit : par défaut on travaille hors ligne, et l'agent d'IA propose tandis qu'une personne décide. Cette règle est une donnée que l'organisation attache à son propre modèle de domaine. Ce modèle se juge-t-il lui-même, avec en lui un acteur LLM qui détient un effet ?

HARO

```
oRule = new stzGraphRule("no-llm-effectful")
oRule.SetDomainQ("agentic")
oRule.WhenQ("kind", "equals", "llm_actor")
oRule.WhenQ("capabilities", "contains", "effectful")
oRule.ThenViolationQ("an LLM proposes, only a pi-gate commits")
g = new stzGraph("agents")
g.AddNode("writer")
g.SetNodeProperty("writer", "kind", "llm_actor")
g.SetNodeProperty("writer", "capabilities", [ "inference", "effectful" ])
g.AddRule(oRule)
? g.RulesAreSound()
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

`RulesAreSound()` répond `0` : le modèle de domaine trouve son propre `writer`, un acteur LLM qui détient `effectful`, en violation de la règle.

Les trois cas sont un restaurateur qui n'est pas programmeur, une banque sous un régulateur et une organisation qui a choisi la possession plutôt qu'un éditeur. Le restaurateur et l'organisation sont des artisans du logiciel selon le dossier ; la banque est titulaire d'une licence d'un produit, et le dossier ne dit pas qui, là-bas, a déclaré quoi que ce soit. Le restaurateur et l'organisation n'ont pas attendu l'industrie du logiciel, et chacun se voit offrir la possession de ce qui est construit pour lui.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai gardé le programmeur en premier. La vision de la plateforme, que j'ai ratifiée, le dit en une ligne : le premier public reste le programmeur d'une petite équipe ; l'ère agentique ne l'a pas remplacé, elle l'a multiplié. J'ai tenu que les agents d'IA ne rendent pas les programmeurs obsolètes.

J'ai ajouté l'agent d'IA comme artisan du logiciel, et retenu un pouvoir. L'agent est le sixième lecteur de chaque langage Softanza : à la fois un constructeur et un utilisateur. Il reçoit un atelier et un tribunal. Il ne reçoit pas la capacité de valider.

J'ai refusé « maker », et j'ai refusé « builder ». « Maker » nomme le mouvement des ateliers, le fer à souder et l'imprimante 3D, comme la connaissance générale le tient. L'accroche du site l'a employé jusqu'au 8 octobre 2026. « Builder » nomme quelqu'un qui assemble ce qu'un plan dit. Aucun des deux ne dit ce que je crois qu'est la programmation. C'est un travail de la main, fait avec soin, sur quelque chose qui est à vous : c'est de l'artisanat. J'appelle la personne qui le fait un artisan du logiciel.

J'ai refusé de compter un client sans document. Le site ne liste un projet comme référence que lorsqu'un document le prouve, et ne cite aucun nombre de pays ni aucun nombre de programmeurs. Le nom d'un client va sur le site sur ma parole ; son rang de référence demande le document. Une leçon n'est listée que lorsqu'un fichier de la bibliothèque la porte.

J'ai refusé de présenter la bibliothèque par son anatomie. Depuis 2022, la porte d'entrée est qui vous êtes et ce que vous voulez faire, et l'anatomie est placée sous chaque étape. Le 9 octobre 2026, le site gardait sept portes, et les parcours derrière elles existaient comme pages, neuf ce jour-là, chaque étape portant son mot d'étape. Que chaque étape s'exécute dans l'ordre sous une garde est conçu ; le construire est l'étape suivante.

Les gardes qui prouvent ces affirmations, et celle qui est en développement, sont dans la section suivante.

5. LE TRIBUNAL

`agentfile_narrated` prouve qu'un agent est un fichier jugé quand il est chargé : un refus par barrière, la couverture manquante parmi eux, et l'agent dont l'esprit est un modèle de langage, qui détient un effet refusé par la même règle que celle qu'emploie le graphe. `service_registry_narrated` prouve qu'un bac à sable en production est une violation. `graphrule_object_narrated` prouve la règle comme un objet attaché à un modèle. `semantic_resolver_narrated` tient le chemin haoussa comme dernier scénario. À ce commit, elle est en développement : son exécution s'arrête après 29 assertions, avant les assertions haoussa, sur une erreur levée par une fonction de découpage de chaînes, et terminer cette exécution est l'étape suivante. La page compte donc ses assertions et ne revendique pas son verdict ; le bloc haoussa a été exécuté seul. `00_orgchart_narrated` prouve les postes, les liens hiérarchiques et le suivi des vacances de l'organigramme ; le constat BCEAO imprimé plus haut vient d'un fichier voisin, `14_bceao_governance.ring`.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`agentfile_narrated` 88 assertions
base/test/agentic/agentfile_narrated.ring

`service_registry_narrated` 93 assertions
base/test/system/service_registry_narrated.ring

`graphrule_object_narrated` 34 assertions
base/test/graph/graphrule_object_narrated.ring

`semantic_resolver_narrated` 83 assertions
base/test/natural/semantic_resolver_narrated.ring

`00_orgchart_narrated` 10 assertions
base/test/orgchart/00_orgchart_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les agents de code mènent en mains. Ils lisent, cherchent, modifient et exécutent déjà. Ce qui leur manque, selon le site lui-même, c'est un moyen de connaître une interface inconnue sans deviner, de juger leur travail selon des règles et des promesses, et de présenter un changement comme une intention qu'une personne peut relire. Les juges de Softanza sont construits ; la porte entre eux et un agent de code est conçue, et la construire est l'étape suivante.

Le low-code mène en base installée et en tout ce qui l'entoure, comme la connaissance générale le tient. L'organisation sur le terrain garde la suite de son éditeur pour ses comptes, ses documents et ses notifications. La plateforme de l'artisan du logiciel se place à côté des outils du terrain, non à leur place.

Les applications de messagerie mènent là où les artisans du logiciel sont déjà, et le dossier du restaurant n'en fait pas encore son médium. L'outil du restaurateur avant le projet était un tableur ; son propre rapport du 4 juillet 2026 le dit. La ligne qui nomme les applications de messagerie est l'intitulé d'une option du formulaire de cadrage, et non sa réponse. Aucune page du site ne nomme ces applications, et aucun médium conversationnel n'est encore une cible de la plateforme. Le chapitre 25 parle de cette direction.

Et la porte du formateur en est à une étape plus précoce que les autres. Les éditions du cours en français, en arabe et en haoussa sont des brouillons qu'un relecteur n'a pas encore lus, et la première institution à l'adopter reste à venir.

La section suivante dit ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT	le fichier d'agent et son tribunal ; le registre qui refuse un faux en production ; des règles comme données sur un modèle ; les règles d'un régulateur qui jugent un organigramme ; la couche naturelle en haoussa et en arabe
CONÇU	une garde qui échoue quand une étape d'une page de parcours casse, chaque étape exécutée dans l'ordre ; la porte par laquelle un agent de code demande à la bibliothèque de juger son travail
ABSENT	les plateformes des trois clients qui tournent sur le moteur Softanza ; le propre médium d'un artisan du logiciel, l'application de messagerie sur son téléphone, comme cible

Ce qui est construit : le fichier d'agent et son tribunal ; le registre qui refuse un faux en production ; des règles comme données sur un modèle ; les règles d'un régulateur qui jugent un organigramme ; la couche naturelle en haoussa et en arabe.

Ce qui est conçu : une garde qui échoue quand une étape d'une page de parcours casse, chaque étape exécutée dans l'ordre ; la porte par laquelle un agent de code demande à la bibliothèque de juger son travail.

Ce qui est absent : les plateformes des trois clients qui tournent sur le moteur Softanza. Aujourd'hui la plateforme du restaurant est un ensemble de mondes web, chacun un fichier unique ; celle de la banque est une application Python ; le hub de l'organisation est une conception. Leurs déclarations sont en cours d'écriture comme celles de Softanza. Et le propre médium de l'artisan du logiciel, l'application de messagerie sur son téléphone, n'est pas encore quelque chose auquel un monde puisse être déclaré.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous déclarez l'intention dans une bibliothèque qui se lit comme une phrase, et un moteur calcule sous chaque appel. Vous restez le premier public.
L'analyste	Vous déclarez en texte brut les entités, les règles et les flux de votre organisation, et les règles jugent le modèle.
Le concepteur	Vous déclarez une image, une carte ou un écran comme un programme, et la première règle d'une interface peut venir d'un vrai métier : pour un restaurateur, la vitesse de commande.
Le décideur	Vous déclarez votre monde, votre gouvernance, et ce que chaque acteur peut valider, et vous possédez tout, en texte brut.
Le formateur	Vous déclarez un cours dans la langue de vos apprenants, et chaque exercice est vérifié en l'exécutant, selon les mots du site. Les éditions en français, en arabe et en haoussa sont des brouillons qu'un relecteur n'a pas encore lus.
L'agent	Vous vous déclarez dans un fichier : ce que vous couvrez, à quel point vos actes sont réversibles. Vous recevez un atelier et un tribunal. Vous ne recevez pas la capacité de valider.

9. EXERCICE

Déclarez un agent d'IA pour un monde que vous connaissez : une boutique, une classe, un bureau. Écrivez son fichier : ce qu'il couvre, à quel point ses actes sont réversibles, quand il s'exécute, une compétence. Avant de l'exécuter, écrivez la promesse que vous attendez sous la question ci-dessous. Puis exécutez-le.

HARO

```
? StzAgentDeclarationQ(cYourAgent).IsValid()
#--> 1
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Si le tribunal refuse, lisez la règle qu'il cite et changez votre fichier, non le tribunal. Puis faites de votre agent un modèle de langage et donnez-lui un effet, et lisez pourquoi il est refusé. Vous ne trouverez pas le fichier corrigé dans ce livre.

Je vous laisse celui-ci. Prenez le bloc de la règle de graphe de la section 3, celui de l'organisation sur le terrain, et changez une ligne : les capacités de `writer` deviennent `[ "inference" ]`, sans `"effectful"`. Exécutez-le et lisez la dernière réponse.

HARO

```
oRule = new stzGraphRule("no-llm-effectful")
oRule.SetDomainQ("agentic")
oRule.WhenQ("kind", "equals", "llm_actor")
oRule.WhenQ("capabilities", "contains", "effectful")
oRule.ThenViolationQ("an LLM proposes, only a pi-gate commits")
g = new stzGraph("agents")
g.AddNode("writer")
g.SetNodeProperty("writer", "kind", "llm_actor")
g.SetNodeProperty("writer", "capabilities", [ "inference" ])
g.AddRule(oRule)
? g.RulesAreSound()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Comparez-la ensuite à la réponse imprimée dans la section 3, et dites ce que la règle a lu pour donner chacune. La section 3 montre ses deux conditions. Quand vous saurez le dire, vous saurez quelle ligne du modèle de domaine la règle lit.

Le chapitre 3 est la méthode, depuis les premiers principes, et la phrase à laquelle sa méthode revient est celle-ci : vous déclarez un monde dans un langage fait pour lui, et vous gouvernez la façon dont ce monde change.

OÙ SONT LES PREUVES

NARRATION	stz-built-on-real-world-need.md
NARRATION	stz-service-virtualization-code-first-subscribe-later.md
NARRATION	stz-agents-that-cannot-hurt-you-narration.md
NARRATION	stz-rules-as-graph-rules-narration.md
NARRATION	stzOrgChartDoc-BankUseCase.md
SUR LE SITE	offering
SUR LE SITE	customers
SUR LE SITE	forged
SUR LE SITE	coding-agents
DÉCISIONS DE L'ÉTUDE	12.2, 12.12, 12.13, 12.25, 12.29, 12.30

PARTIE I · CHAPITRE 3 · COLONNE

La méthode, depuis les premiers principes

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 5 bloc(s) sur 5 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je montre la méthode que suit chaque chapitre après celui-ci. Je prends « depuis les premiers principes », le slogan du terrain, et je lui donne un sens étroit : un concept, la forme du terrain, une reconception en code qui s'exécute, et un tribunal qui la juge. Je montre le plus petit réexamen que je connaisse, la chaîne vide, puis les deux actes, déclarer et gouverner, chacun en quelques lignes. Je dis ce que j'ai refusé. D'abord une fonction sans besoin, et le code généraliste comme lieu de rencontre entre les personnes et les machines. Ensuite un titre qui regardait en arrière, la grille seule, et une lecture appelée recensement. Toute la méthode revient à une phrase : vous déclarez un monde dans un langage fait pour lui, et vous gouvernez la façon dont ce monde change.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Dans notre industrie, « depuis les premiers principes » est un slogan. Cela veut dire raisonner à partir du bas, non par analogie. Vous l'entendrez dans des conférences et le lirez dans des annonces de lancement, et on vous montrera rarement ce qu'il a produit.

Le terrain a bien une méthode pour les décisions derrière le logiciel, et c'est une bonne méthode. Une équipe écrit une courte fiche pour chaque décision qu'elle prend, composée du contexte, de la décision et des conséquences. Beaucoup d'équipes les appellent des fiches de décision d'architecture, d'après un billet de blog

que Michael Nygard a écrit en 2011. Le terrain a aussi une méthode pour prouver ce que fait une fonction : un exemple écrit sous la fonction, que les outils peuvent exécuter comme un test. Python l'offre depuis 2001 avec son module doctest, qui s'exécute quand quelqu'un l'invoque, et Rust exécute par défaut les exemples de sa documentation comme des tests. Et quand le terrain compare un outil à un autre, il dessine une grille : des lignes de fonctions, une colonne par outil, des coches et des croix.

Le terrain a trois formes ; la méthode revient à une seule phrase.

2. LE PREMIER PRINCIPE

Voici la phrase à laquelle toute la méthode revient : **vous déclarez un monde dans un langage fait pour lui, et vous gouvernez la façon dont ce monde change.**

Deux actes. Tout le reste de ce livre est l'un d'eux, ou les deux ensemble.

3. LA FORME SOFTANZA

« Depuis les premiers principes » peut se vérifier. Dans ce livre, cela a un sens étroit. Prenez un concept que tout le monde utilise, écrivez comment le terrain le tient, reconcevez-le, et laissez un code qui porte la reconception et un tribunal qui la juge. La page d'histoire du site s'intitule *Depuis les premiers principes*, et la bibliothèque est sur GitHub depuis le 12 mars 2022. L'expression n'est pas ici une décoration. C'est un ordre de travail.

Laissez-moi vous montrer le plus petit réexamen que je connaisse. Demandez à Python combien de fois la chaîne vide apparaît dans le mot « text », et il répond cinq : une avant chaque lettre et une à la fin. Demandez-lui si « text » contient la chaîne vide, et il dit oui ; JavaScript dit la même chose. J'ai interrogé Python 3.13 et Node 22, le 8 octobre 2026. C'est une convention d'implémenteurs, non un fait que quiconque dirait à voix haute. Softanza énonce une règle à la place, et la règle a un nom dans la narration qui l'argumente : le vide ne se compte pas, et le vide ne se contient pas. Que répond Softanza aux deux mêmes questions ? Exécutez-le :

```
HARO
```

```
? Q("text").Count("")
#--> 0
? Q("text").Contains("")
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `0` et `FALSE`. Deux lignes, et une décision avec laquelle vous pouvez être en désaccord. Voilà à quoi ressemble un réexamen quand il est honnête : assez petit pour être lu, assez explicite pour être refusé.

Une lecture du dossier, faite en octobre 2026, a compté quarante-six concepts de cette sorte, en sept familles. Les deux premières sont les paradigmes, tels que la programmation orientée portée, la programmation de la connaissance et la programmation réactive reconçus, et les moteurs de paradigme, tels que la vérité, les quantificateurs et le temps. Les cinq autres sont les patrons et la réaction ; l'expression et la perception ; les données, le temps et le raisonnement ; le système, la livraison et les opérations ; et la bibliothèque qui se connaît elle-même. Quarante-six est une lecture et non un recensement ; la page de registre du site lui-même, lue le 9 octobre 2026, dit quarante-sept. Chaque concept de la liste a son propre article ou document de conception, et c'était le critère pour entrer dans la liste.

Voyons maintenant les deux actes, chacun en quelques lignes.

Déclarer. Une sorte de valeur est déclarée en une phrase, et chaque verdict à son égard est calculé et s'explique lui-même. Pourquoi -4 ne peut-il pas être un score ?

HARO

```
ConstrainQ("score").ToBeQ(:Number).AndQ().ToBeQ(:Positive).AndQ().ToBeQ(:Even)
? CanQ(42).BeA("score")
#--> TRUE
? CanQ(-4).BeA("score")
#--> FALSE
? Why()
#--> no: -4 is not positive (constraint 2 of 'score')
```

✓ exécuté : a affiché ce que ce bloc promet

Remarquez la dernière ligne : `no: -4 is not positive (constraint 2 of 'score')`. Le verdict nomme la règle qu'il a appliquée, par son numéro ! Un verdict sur une sorte déclarée qui ne peut pas dire pourquoi n'est pas admis dans cette maison.

Gouverner. Une contrainte déclarée, une fois imposée, refuse un changement avant qu'il atterrisse. Ici la chaîne « hello » doit rester en minuscules. La mise à jour vers « HELLO » atterrit-elle ?

HARO

```
oName = new stzString("hello")
oName.EnforceConstraint("keep-lowercase", :Lowercase)
bRefused = FALSE
try
        oName.UpdateWith("HELLO")
catch
        bRefused = TRUE
done
? bRefused
#--> TRUE
? oName.Content()
#--> hello
```

✓ exécuté : a affiché ce que ce bloc promet

La mise à jour a été refusée ! L'objet reste comme il était : il répond encore `hello`. Déclarer une contrainte et l'imposer sont deux actes, et la garde derrière ce bloc vérifie qu'une contrainte seulement déclarée reste passive.

Gouverner a aussi un côté qui fait face à la personne. Le système demande, et il ne vous laisse pas clore un monde qui n'est pas complet. Que dit-il quand vous essayez de conclure quand même ?

HARO

```
oKB = new stzKnowledgeGraph("restaurant")
oKB {
        Know("margherita", "dish")
        Know("tiramisu", "dish")
        AddConversationQ("setup").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        aQ = AskInXT("setup")
        ? aQ[:why]
```

```
        #--> every dish needs 'contains'
}
try
        oKB.ConcludeIn("setup", "world")
catch
        ? cCatchError
done
#--> Can't conclude: 2 gap(s) remain -- the wise-coding loop is not done. Ask the next question.
```

✓ exécuté : a affiché ce que ce bloc promet

Le but a été déclaré : chaque plat doit dire ce qu'il contient. L'écart entre le but et ce qui est connu génère la question, et la question porte sa raison. Conclure alors que deux écarts demeurent est refusé ! La ligne imprimée est `Can't conclude: 2 gap(s) remain -- the wise-coding loop is not done. Ask the next question.`, et ses derniers mots vous disent quoi faire ensuite. Le chapitre 18 parle de cette conversation. Ici il suffit de voir que gouverner n'est pas seulement refuser ; c'est demander.

Encore un réexamen, parce que les moteurs de paradigme font aussi partie de la méthode. La vérité, dans le langage sous Softanza, est une affaire de vide et de non-vide, et une liste de mots compte comme vraie. Softanza garde cette vérité par défaut et vous laisse l'étendre par déclaration. Une fois « wrong » déclaré comme un élément qui rend une liste fausse, une liste qui le contient est-elle encore vraie ?

HARO

```
SetItemsMakingAListFalse([ "false", "wrong", "dangerous" ])
? IsTrue([ "hello", "this", "is", "wrong" ])
#--> TRUE
? IsTrueXT([ "hello", "this", "is", "wrong" ])
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

La réponse de l'hôte est inchangée : `TRUE`. La réponse étendue lit votre déclaration et dit `FALSE` ! « La vérité devient une affaire de politique », dit la narration, et la politique est écrite là où vous pouvez la lire.

Voilà donc la méthode, sous la forme que vous rencontrerez dans chaque chapitre après celui-ci. Un concept tel que le terrain le tient. Le seul principe auquel le dossier revient. La reconception, en code qui s'exécute, avec sa promesse sous chaque bloc. La décision, et ce qui a été refusé. Le tribunal qui la prouve, par son nom. Le terrain, et où il mène. La limite, en trois mots d'étape. Le premier, **construit**, veut dire que le code existe, qu'une garde existe et qu'un bloc a tourné ; c'est le *construit* du site lui-même. Le deuxième, **conçu**, couvre ce que le site appelle *en construction*, *spécification* et *proposition ratifiée*. Le troisième, **absent**, couvre ce que le site appelle *nommé*, un nom sans rien derrière lui pour l'instant, et ce que personne n'a nommé. Cinq mots du site, trois du livre. Puis six portes, une pour chaque sorte de lecteur, et un exercice.

Et le livre est jugé comme la bibliothèque l'est. Chaque bloc qu'il contient s'est exécuté dans la bibliothèque la nuit où la page a été construite, au commit imprimé en haut de la page. Une page dont les promesses ne sont pas tenues ne se construit pas.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Il n'existe pas de fiche unique d'une décision appelée « la méthode ». La méthode a été décidée par morceaux, et les morceaux sont dans le dossier.

J'ai refusé d'ajouter une fonction sans besoin. Je l'ai écrit dans un article de la bibliothèque, qui ne porte pas de date. « Je n'introduis de nouvelles fonctions que lorsqu'elles résolvent un problème tangible ou lorsque j'anticipe qu'elles répondront aux besoins des programmeurs. » Les premiers principes sans cette règle produisent un musée. Avec elle, un réexamen doit mériter sa place : quelqu'un avait un problème auquel la forme du terrain ne répondait pas.

J'ai refusé le code généraliste comme lieu de rencontre entre les personnes et les machines. La vision de la plateforme le dit en une ligne, que j'ai ratifiée. Les humains et les IA se rencontrent dans des langages déclarés, petits, clos et jugés, et non dans du code généraliste. Déclarer est le premier acte parce qu'il rend le second possible. Un tribunal ne peut juger que la phrase d'un langage qu'il connaît.

J'ai refusé un titre qui regardait en arrière. Quand ce livre a été proposé pour la première fois, son titre était *Programming by Heart*, le slogan de la présentation de 2020. Je l'ai écarté comme dépassé : il nommait une posture, et le livre devait nommer ce qui était repensé. La proposition suivante était *Declare, Then Govern*. La couverture porte à la place la phrase que donne la préface : déclarez ce que vous savez, un tribunal le garde, le modèle d'IA vient en dernier. Les deux verbes sont passés en quatrième de couverture, où vous les trouverez. Ils organisent chaque chapitre ; ils n'avaient pas besoin d'en être le nom.

J'ai refusé la grille seule. Les documents de la bibliothèque contiennent des dizaines de grilles de comparaison, et dans celles-ci la colonne de Softanza gagne presque chaque ligne. Un lecteur qui ne voit que ces grilles a raison d'en demander davantage qu'une grille. L'honnêteté du dossier est ailleurs : dans les sections des études de terrain intitulées « ce que le terrain a et qui manque à Softanza,

honnêtement », et dans des chiffres mesurés. Quand le runtime sous Softanza a été mesuré de nouveau en septembre 2026 contre celui auquel il était comparé, le résultat annoncé auparavant ne s'est pas reproduit. La perte a été consignée le jour même, dans une note interne. C'est la règle que suit ce livre : une perte est imprimée à côté du gain, dans la même section.

J'ai refusé d'appeler une lecture un recensement. Les quarante-six concepts sont une lecture du dossier par une session, non une liste que j'ai tranchée ligne par ligne. L'annexe A imprime le registre de quarante-six lignes tel que l'étude le donne, avec la colonne propre au livre, comme annexe de travail : les lignes n'ont pas été tranchées une à une.

Voilà les cinq refus ; voyons maintenant le tribunal qui juge les blocs ci-dessus.

5. LE TRIBUNAL

Les trois blocs des deux actes reposent sur deux gardes. `modality_narrated` prouve qu'une sorte déclarée peut être inspectée, qu'un verdict nomme la contrainte qui a échoué, et qu'une sorte inconnue n'autorise jamais en silence. Elle prouve aussi qu'une contrainte imposée refuse une mauvaise mise à jour avant qu'elle atterrisse, tandis qu'une contrainte déclarée reste passive. `wisecoding_narrated` prouve la conversation. Elle prouve des écarts calculés à partir du but déclaré, une question qui dit pourquoi elle est posée, des réponses admises, et une loi qui refuse à travers la conversation et atteint une personne. Et elle prouve que conclure avec des écarts ouverts est refusé. Sa sœur, `wisecoding_rich_narrated`, porte les cas plus riches de la même boucle.

Le réexamen du vide repose sur `612_dealing_in_emptiness_in_softanza.ring`, un test de la bibliothèque écrit en scénarios, chaque ligne énonçant le résultat qu'elle attend. Le réexamen de la vérité repose sur sa narration et sur l'exécution de ce livre, et sa garde est l'étape suivante. Le fichier de la bibliothèque qui lui est consacré, `04_truth_in_softanza.ring`, fixe la liste des mots faux par affectation et n'appelle jamais la fonction, de sorte qu'il ne prouve pas le bloc.

Le tribunal est nommé ; vient ensuite le terrain, et où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`modality_narrated` 45 assertions
base/test/natural/modality_narrated.ring

`wisecoding_narrated` 13 assertions
base/test/conversation/wisecoding_narrated.ring

`wisecoding_rich_narrated` 52 assertions
base/test/conversation/wisecoding_rich_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en adoption. Les fiches de décision sont un standard léger que beaucoup d'équipes utilisent, avec des modèles et des outils autour. L'élément décision de ce livre existe dans ce livre et dans une étude du dossier, et pas encore comme donnée dans la bibliothèque. Les exemples de la documentation peuvent être exécutés par l'outil standard de Python, et Rust les exécute par défaut. Les promesses de Softanza sont jugées par les exécuteurs propres de la maison, qu'aucun outil extérieur ne lit encore. Et le terrain compare par grille, comme les pages du dossier le font aussi ; une grille seule est un début, et c'est pourquoi la section 4 demande la perte à côté du gain.

Là où je pense que Softanza se distingue, c'est en un seul endroit : le réexamen est mené jusqu'au bout. Je tiens cela pour une vue, non pour un résultat. La décision est écrite, la reconception s'exécute, la promesse est jugée à la construction, et le mot d'étape dit jusqu'où chaque pièce est allée. Une fiche de décision ne s'exécute pas. Un exemple ne dit pas ce qui a été refusé. Ici, ils sont un seul objet.

7. LA LIMITE

CONSTRUIT	le tribunal derrière chaque bloc de ce chapitre sauf celui de la vérité, et les mots d'étape que le site imprime sur chaque produit
CONÇU	le registre des concepts repensés comme données, avec une garde qui échoue quand un concept n'a pas de preuve
ABSENT	une page qui montre chaque concept repensé avec une preuve exécutée ; un fichier de tests pour le moteur de paradigme de la fusion de sections

Ce qui est construit : le tribunal derrière chaque bloc de ce chapitre sauf celui de la vérité, et les cinq mots d'étape que le site imprime sur chaque produit.

Ce qui est conçu : le registre des concepts repensés comme données, une ligne par concept avec son article, son code, sa garde et son étape. À côté, une garde qui échoue quand un nouveau concept arrive sans ligne. Les deux sont aujourd'hui une spécification.

Ce qui est absent : une page qui montre chaque concept avec une preuve exécutée ; la page de registre du site, lue le 9 octobre 2026, les liste chacun avec ses preuves. Un fichier de tests pour le module du moteur de l'un des moteurs de paradigme, la fusion de sections : il n'en a pas en propre, bien que ses méthodes de fusion s'exécutent dans `base/test/string/289_content.ring` et `max/test/stzListOfPairsTest.ring`. Et quelques réexamens où l'article va un pas au-delà du code d'aujourd'hui. L'article sur les quantificateurs demande si toutes vos notes `AreGreaterThan` la moyenne, et cette méthode s'écrit autrement dans la bibliothèque aujourd'hui. Ses quantificateurs plus conviviaux, certains, la plupart et peu, tirent au hasard, de sorte qu'ils ne portent aucune promesse aujourd'hui. Le registre est une lecture, et une lecture peut manquer un concept qui ne vit que dans un seul article.

Là s'arrête ce chapitre ; six portes suivent, une pour chaque sorte de lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Chaque affirmation que vous faites peut porter sa promesse. Une contrainte imposée refuse une mauvaise mise à jour au point de mise à jour unique de l'objet, celui que `UpdateWith()` atteint : `Update()` pour une chaîne ou un nombre, `_SetContent()` pour une liste.
L'analyste	Votre règle est une déclaration, et son verdict nomme la règle qui a échoué, par son numéro.
Le concepteur	Le mot d'étape fait partie de l'interface. Un produit qui n'existe pas encore le dit, en un mot que vous pouvez vérifier.
Le décideur	Demandez à chaque fournisseur la perte à côté du gain. Le chapitre 1 en imprime une, avec sa date : le runtime sous Softanza est derrière celui auquel il était comparé, d'un facteur entre un et deux sur chaque ligne, mesuré le 13 septembre 2026 ; le chapitre 27 donne les lignes.
Le formateur	Chaque réexamen a un article qui l'argumente : le concept, la forme du terrain, la reconception. C'est un programme de cours, ordonné par famille.
L'agent	Vous apprenez cette plateforme à partir de la plateforme, et elle vous corrige. Un verdict qui dit pourquoi est écrit pour vous autant que pour une personne.

9. EXERCICE

Choisissez un défaut de votre propre langage que vous n'avez jamais questionné : ce que compte la chaîne vide, ce qui compte comme vrai, ce que veut dire « la plupart ». Avant d'exécuter quoi que ce soit, écrivez en une phrase la règle que vous croyez juste. Puis écrivez deux promesses sous la forme que vous avez rencontrée : la ligne que votre langage imprime aujourd'hui, et la ligne que votre règle doit imprimer.

HARO

```
? YourDefault(the_input)
#--> what your language prints today
? YourRule(the_input)
#--> what your declared rule must print
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez les deux. Je ne vous dirai pas quelle règle est juste. Si votre règle et votre langage divergent, vous avez trouvé une décision que quelqu'un a prise pour vous, et vous pouvez maintenant la prendre vous-même.

Je vous laisse celui-ci, volontairement. Reprenez le bloc de la section 3 qui gouverne une chaîne, changez un seul mot, `EnforceConstraint` pour `AddConstraint`, et exécutez-le tel qu'il est donné ci-dessous. Je n'en imprime aucune sortie.

HARO

```
oName = new stzString("hello")
oName.AddConstraint("keep-lowercase", :Lowercase)
bRefused = FALSE
try
        oName.UpdateWith("HELLO")
catch
        bRefused = TRUE
done
? bRefused
? oName.Content()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Regardez ce qu'il imprime : si la mise à jour a été refusée, et ce que l'objet contient. Lisez ensuite la section 3, juste après le bloc, et la section 5, sur la garde qui le vérifie : la page dit pourquoi. Quand vous aurez les deux lignes, vous saurez pourquoi une contrainte déclarée n'est pas une contrainte imposée.

Le chapitre 4 dessine la plateforme en une image, sous une seule idée à chaque altitude : rien n'est cru, tout est jugé.

OÙ SONT LES PREUVES

NARRATION	`stzstring-emptiness-narration.md`
NARRATION	`istrue-softanza-truth.md`
NARRATION	`stz-built-on-real-world-need.md`
SUR LE SITE	history
SUR LE SITE	principles
SUR LE SITE	estate
SUR LE SITE	compare
DÉCISIONS DE L'ÉTUDE	12.19, 12.22, 12.29, 12.30

PARTIE I · CHAPITRE 4 · COLONNE

La plateforme en une image

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 8 bloc(s) sur 8 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je dessine toute la plateforme en une image, lue de bas en haut : le moteur, les couches, les zones, les mondes, les artisans du logiciel, et à côté de chaque étage son tribunal. Je la parcours étage par étage, avec à chaque étage un bloc que vous pouvez exécuter. Puis je dis ce que j'ai refusé en chemin : le calcul dans les faces, un drapeau pour choisir une couche, les mini-applications. L'image se résume en une seule idée à chaque altitude : rien n'est cru, tout est jugé.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain dessine une plateforme comme une pile de boîtes. Le noyau en bas, un runtime et ses bibliothèques au milieu, les applications au-dessus, et parfois une colonne sur le côté pour la sécurité ou les outils. À l'intérieur d'une même base de code, la même image devient une règle sur les dépendances : une couche interne ne sait rien des couches externes, et les flèches ne pointent que vers l'intérieur. Les trois crates standard de Rust, une pour le métal nu, une pour la mémoire, une pour un système d'exploitation complet, en sont la version la plus nette que je connaisse. Trois profondeurs du même langage, choisies par ce que la cible offre.

Pour les applications, l'image du terrain est un hôte. Une grande application embarque un runtime et fait tourner de petites applications à l'intérieur, qui partagent ses services et son magasin. Dans les deux images, le juge est à l'extérieur : une suite de tests à côté du code, une revue à côté du magasin. Dans les images que je connais, aucune ne dessine le tribunal comme une colonne debout à chaque étage.

Softanza dessine cette colonne, et une seule idée tient toute l'image.

2. LE PREMIER PRINCIPE

La pile de la plateforme se résume, dans la vision que j'ai ratifiée, en une seule idée à chaque altitude : **rien n'est cru, tout est jugé.**

3. LA FORME SOFTANZA

Voici toute la plateforme en une image, lue de bas en haut. Chaque étage repose sur celui qui est dessous, et chaque étage est jugé par un tribunal à côté de lui. Les chiffres qu'elle porte ne sont pas tapés par moi. Le build les lit, quand cette page est faite, dans la bibliothèque et dans la carte des zones du site.

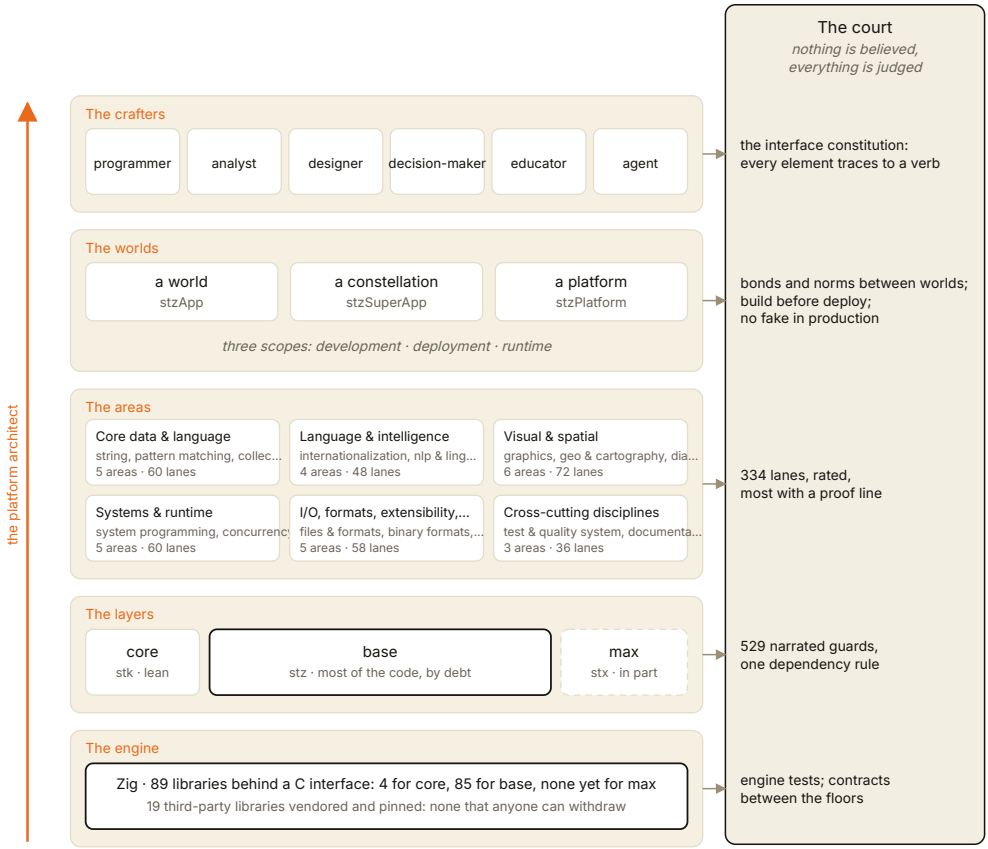


Fig. *La plateforme en une image. Chaque étage repose sur celui qui est dessous ; chacun est jugé par le tribunal à côté de lui ; l'architecte travaille à travers tous. Les chiffres sont lus au build dans le moteur de la bibliothèque, dans son arbre de tests et dans l'Atlas du site.*

Parcourons-la, étage par étage, avec à chaque étage un bloc que vous pouvez exécuter.

Le moteur. En bas se trouve un moteur écrit en Zig : un ensemble de bibliothèques derrière une interface C, une par domaine, regroupées par la couche qui les charge. Selon la règle de conception du moteur, le moteur fait le calcul ; la bibliothèque au-

dessus lui donne un vocabulaire. Un petit exemple rend la division visible. Demandez au langage sous-jacent à Softanza la longueur du mot arabe pour la paix, puis demandez-la au moteur. Que compte chacun ? Exécutez-le :

HARO

```
? len("سلام")
#--> 8
? Q("سلام").NumberOfChars()
#--> 4
? Q("مرحبا بالعالم").Script()
#--> arabic
```

✓ exécuté : a affiché ce que ce bloc promet

Le langage compte des octets et répond `8`. Le moteur compte des lettres et répond `4`, et il connaît l'écriture : `arabic`. Quatre lettres, pas huit !

Le moteur a aussi sa propre porte, sous chaque classe. Les nombres trop grands pour l'arithmétique de la machine sont additionnés exactement, par le moteur lui-même :

HARO

```
a = StzEngineBigIntFromString("99999999999999999999999999999999")
? StzEngineBigIntToString( StzEngineBigIntAdd(a, StzEngineBigIntFromString("1")) )
#--> 100000000000000000000000000000000
```

✓ exécuté : a affiché ce que ce bloc promet

Le moteur répond `100000000000000000000000000000000`, la somme exacte.

Les couches. Au-dessus du moteur, la bibliothèque est découpée en trois couches, selon la profondeur de capacité. Le noyau, avec le préfixe `stk`, est léger et parle directement au moteur, pour les petits appareils et les petits binaires. La base, `stz`, est la couche riche, où vivent le vocabulaire et ses formes, assez pour la plupart des solutions. Le max, `stx`, est destiné aux innovations les plus fortes. Une couche se choisit par le fichier qu'un programme charge. Le même verbe veut-il dire le même acte au noyau et à la base ? Exécutez-le :

HARO

```
o1 = new stkList([ "one", "and", "one", "makes", "two", "ones" ])
? o1.FindFirst("one")
#--> 1
o2 = new stzList([ "one", "and", "one", "makes", "two", "ones" ])
? o2.FindFirst("one")
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux répondent `1` : le même verbe, le même acte, à deux profondeurs.

Les zones. À travers les couches, la bibliothèque est cartographiée par l'Atlas du site en zones regroupées en bandes, et chaque zone en voies. Chaque voie est notée d'émergente à forte, la plupart avec la ligne qui le prouve. L'Atlas dit ce qui existe, et à quel point c'est bien. Un autre principe de la conception du moteur est qu'un verbe veut dire le même acte sur toute structure qui l'admet. Tient-elle pour une paire, une chaîne et une liste ? Exécutez-le :

HARO

```
? @@( Q("tea rice tea").FindAll("tea") )
#--> [ 1, 10 ]
? @@( Q([ "tea", "rice", "tea" ]).FindAll("tea") )
#--> [ 1, 3 ]
```

✓ exécuté : a affiché ce que ce bloc promet

Sur la chaîne, `FindAll` répond des positions de caractères : `[ 1, 10 ]`. Sur la liste, il répond des positions d'éléments : `[ 1, 3 ]`. Le même acte, sur deux structures.

Les mondes. Au-dessus de la bibliothèque, l'unité n'est pas un module mais un monde : une application est un monde vivant de sens, avec ses choses, ses relations et ses règles. Un monde se déclare. Que dit-il de lui-même quand on lui demande de s'expliquer ? Exécutez-le :

HARO

```
oApp = new stzApp("FieldVisits")
oApp {
```

```
    AddThingQ(:Account) { Has([ :number, :chapter, :balance ])  IsTrue(:balance, "@ >= 0") }
    AddThingQ(:Client)  { Has([ :code, :name, :city ])  Owns(:Account) }
    AddRelation(:Account, :belongsTo, :Client)
}
oApp.Explain()
#--> WORLD FieldVisits   lives in: memory (not persisted)
#      BEING
#        account (number, chapter, balance)
#            true when balance @ >= 0
#        client (code, name, city)
#      RELATIONS
#        client owns account
#        account belongsto client
oApp.Live()
#--> [FieldVisits] is live -- 2 thing(s), 0 flow(s), 0 reaction(s), 0 goal(s); 0 proposal(s)
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `WORLD FieldVisits   lives in: memory (not persisted)`, puis liste son être et ses relations. `Live()` répond `[FieldVisits] is live -- 2 thing(s), 0 flow(s), 0 reaction(s), 0 goal(s); 0 proposal(s)`. Deux choses, et ni flux, ni réaction, ni but. Le monde dit ce qui a été déclaré, et rien de plus.

Les mondes se composent en constellation. Ici un restaurant et son fournisseur de produits sont deux mondes, et un lien entre eux déclare l'unique traversée permise. Les trois lignes `Gov` déclarent le risque de la traversée, l'accordent au restaurant et fixent l'autorité du restaurant ; la garde derrière ce bloc montre que sans elles, même la traversée déclarée est refusée. Que devient une traversée qu'aucun lien ne déclare ? Exécutez-le :

HARO

```
oResto = StzAppQ("resto")
oResto.AddThingQ(:dish) { Has([ :name, :price ]) }
oSupplier = StzAppQ("supplier")
oSupplier.AddThingQ(:crate) { Has([ :produce, :qty ]) }
oCon = new stzSuperApp("acme-group")
oCon {
        AddWorld("resto", oResto)
        AddWorld("supplier", oSupplier)
        Bond("resto", "supplier", "order-produce")
        GovDeclareRisk("order-produce", 2)
        GovGrant("resto", "order-produce")
        GovSetAuthority("resto", :Delegated)
```

```
        ? CallAcross("resto", "supplier", "order-produce")
        #--> 1
        ? CallAcross("resto", "supplier", "raid-kitchen")
        #--> 0
        ? Why()
        #--> no bond declares 'raid-kitchen' from 'resto' to 'supplier'.
}
```

✓ exécuté : a affiché ce que ce bloc promet

La traversée déclarée répond `1`. L'autre répond `0`, refusée ! La constellation donne sa raison en une ligne : `no bond declares 'raid-kitchen' from 'resto' to 'supplier'.` Toute autre traversée est refusée avec sa raison.

Le socle commun est le sol partagé d'une constellation, ce sur quoi chacun de ses mondes se tient. Il tient une identité, une session et un magasin. C'est un monde comme les autres, le monde zéro, et non un genre particulier de nœud. Le chapitre 23 le montre et dit jusqu'où le code est allé dans cette direction.

Et une plateforme, `stzPlatform` dans la figure, est l'enveloppe autour d'une solution : ses parties, les systèmes sur lesquels elles tournent, et l'ordre dans lequel elle est construite, répétée et déployée. Vous déclarez la solution qu'elle porte comme un `stzPlatformProfile`, et vous la verrez refuser de déployer avant d'être construite au chapitre 33. Voici ce que son planificateur de livraison, `stzDelivery`, fait des couches, pour le web : une partie d'une solution ne livre que les groupes du moteur dont elle a besoin, et seulement ceux qui appartiennent à Softanza. Le téléphone ci-dessous demande quatre capacités. Combien de groupes du moteur livre-t-il ? Exécutez-le :

HARO

```
oDelivery = new stzDelivery("my-shop")
oDelivery {
        AddSuperApp(:phone, :Android)
        AddBackend(:api, :LinuxServer)
        NeedsIn(:phone, [ :Unicode, :PivotTable, :ConstraintSolver, :Regex ])
        oPlan = Plan()
}
? @@( oPlan.EngineCapsFor(:phone) )
#--> [ "pivottable", "constraintsolver" ]
```

```
? @@( StzWasmGroupsFor( oPlan.EngineCapsFor(:phone) ) )
#--> [ "aggregation", "solver" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Deux, pas quatre : `aggregation` et `solver` ! Le système propre du téléphone gère l'une des deux autres, Unicode, et Softanza livre l'autre, Regex, comme une construction dans le langage propre de la cible.

Les artisans du logiciel. En haut se trouvent les six portes du chapitre 2 : le programmeur, l'analyste, le concepteur, le décideur, le formateur et l'agent. Et une personne travaille à travers chaque étage, du moteur à l'organisation : l'architecte de plateforme. Le dossier nomme ce rôle dans trois de ses documents, et aucun autre rôle ne traverse toutes les bandes de l'Atlas.

Le tribunal. À côté de chaque étage se tient son juge. Le moteur a ses tests et les contrats entre les étages. Les couches ont les gardes narrées. La plupart des voies des zones portent une ligne de preuve. Les mondes ont leurs liens et leurs normes, l'ordre de construction et de déploiement, et le refus d'un faux en production. Les interfaces des artisans du logiciel ont une constitution de verbes et de règles à laquelle chaque élément doit remonter. Et au-dessus de tout cela, le modèle de menaces de la plateforme énumère ses garanties, chacune liée à la garde qui la prouve. Sa propre garde échoue quand une ligne nomme une garde qui n'existe plus.

Les parties II à IV de ce livre sont des repensées, et chacune est une boîte de cette image. La partie V assemble l'image en usage.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé le calcul dans les faces. Le moteur est le produit. Le langage au-dessus ajoute de la beauté, pas du calcul : des milliers de méthodes qui se lisent comme des phrases, et les algorithmes en dessous, en un seul endroit. La règle est que chaque fonctionnalité est implémentée une fois, dans le moteur. La migration est en développement. Le contrat des couches liste le nombre, la liste, l'objet, le graphe, les statistiques et le naturel comme encore dans le langage hôte pour lequel la bibliothèque a d'abord été écrite.

J'ai retiré l'ancien framework, sans repli. La bibliothèque reposait sur un grand framework C++ pour son texte, ses dates et ses fichiers. Quand le moteur l'a remplacé, tous les chemins passant par le framework ont été retirés : pas de repli, pas de drapeau de fonctionnalité gardé au cas où.

J'ai choisi la souveraineté comme un verdict sur chaque dépendance, non comme « chaque ligne est à nous ». Le moteur embarque et épingle un ensemble de bibliothèques tierces, la figure les compte. Le dossier définit le mot en une ligne : rien que quelqu'un d'autre puisse retirer. C'est le test. Une bibliothèque dont la source est épinglée dans l'arbre le passe.

J'ai refusé un drapeau pour choisir une couche. La première manière de choisir une couche était un drapeau. Elle ne pouvait jamais fonctionner, parce que le langage sous-jacent charge à la compilation tous les fichiers qu'il voit, si bien que les trois couches se chargeaient à chaque appel. L'échec de l'une était l'échec de toutes. Le drapeau a été retiré plutôt que gardé comme un interrupteur qui ne fait rien en silence. La raison a été écrite là où le prochain lecteur se tiendra : une couche se choisit par le fichier que l'on charge.

J'ai laissé tout aller dans la base, et je le dis. Sous la pression du temps, les domaines ont atterri dans la base. Le moteur, les volets comme la sécurité et la performance, et les clients avançaient pendant ce temps. C'était le choix agile, et

l'étape suivante est de placer chaque domaine dans sa couche, à la vue de tous. Alors la figure dessine la base au plus large, et le chapitre 27 en donne les chiffres.

J'ai refusé les mini-applications. La super-application du terrain est un hôte qui fait tourner de petites applications. Celle de Softanza est un graphe gouverné de mondes : des mondes, pas des mini-applications ; des liens et des normes, pas un conteneur.

Cinq gardes se tiennent à côté des étages ; la section suivante dit ce que chacune prouve.

5. LE TRIBUNAL

`00_stzapp_narrated` prouve qu'un monde est déclaré et rendu visible. `06_stzsuperapp_narrated` prouve la constellation : des mondes qui se joignent, le socle commun comme monde zéro, des traversées barrées par des normes avec le lien nommé comme raison, un monde retiré qui rompt ses liens. `full_scope_model_narrated`, le terrain d'entente de l'architecte, prouve la solution modélisée d'abord, les portées qui refusent ce qu'une cible interdit, et l'ordre de construction et de déploiement. `01_platform_narrated` prouve les devoirs de la plateforme. `threat_model_narrated` lit le modèle de menaces de la plateforme et échoue quand une garantie nomme une garde qui n'existe plus.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`00_stzapp_narrated` 3 assertions
base/test/app/00_stzapp_narrated.ring

`06_stzsuperapp_narrated` 21 assertions
base/test/app/06_stzsuperapp_narrated.ring

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

`01_platform_narrated` 31 assertions
base/test/platform/01_platform_narrated.ring

`threat_model_narrated` 11 assertions
base/test/security/threat_model_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les trois profondeurs de Rust sont imposées par son compilateur et employées chaque jour sur le métal nu. La règle de Softanza est tenue au moteur et aux fichiers d'entrée, et pas encore portée là où la base devrait s'appuyer sur le noyau, comme le montre la section 7.

Les bibliothèques de texte du terrain mènent sur la locale : la propre carte de Softanza nomme la couche de locale des composants internationaux comme l'écart net de ses chaînes, et nomme le pari derrière lui : sa zone de locale laisse ces composants dehors à dessein, sans collation, sans règles de pluriel et sans fuseaux horaires.

À l'échelle de vrais utilisateurs, les hôtes de super-applications du terrain mènent aujourd'hui. Le dossier les nomme comme le modèle qu'il a refusé et ne mesure rien contre eux, alors je n'en mesurerai pas non plus.

Et l'Atlas lui-même imprime ce qui reste à venir pour Softanza. La zone binaire est pour l'instant un plan, et son code livré est l'étape suivante ; le volet de l'extensibilité (greffons, interface de fonctions étrangères, embarqué) est surtout prévu ; la zone du système compte peu de voies notées fortes à ce jour. Je tiens cela pour une vue, non pour un résultat. Une carte qui imprime ses étapes est la seule sorte qui vaille la peine d'être lue.

7. LA LIMITE

CONSTRUIT — le moteur et ses bibliothèques ; la couche base ; les mondes, les constellations et les plateformes avec leurs gardes ; le sous-ensemble du moteur qu'une partie livre, pour le web

CONÇU — la couche max chargée par son fichier d'entrée ; la couche base s'appuyant sur la couche noyau ; un chargeur pour un seul domaine de la base ; un tribunal qui vérifie les couches contre leur contrat

ABSENT — un programme entier et son moteur emballés dans un seul fichier ; une plateforme cliente sur le moteur

Je mets maintenant sur l'image les trois mots d'étape, construit, conçu et absent.

Ce qui est construit : le moteur et ses bibliothèques, la couche base, les mondes, les constellations et les plateformes avec leurs gardes, et, pour le web, le sous-ensemble du moteur qu'une partie livre.

Ce qui est conçu, ou construit et inaccessible. Les propres classes de la couche max sont écrites et chargées par aucun fichier d'entrée. Que répond le runtime quand vous en demandez une ? Exécutez-le :

HARO

```
try
        o3 = new stxString("one and one")
catch
        ? cCatchError
```

```
done
#--> Error (R11) : Error in class name, class not found: stxstring
```

✓ exécuté : a affiché ce que ce bloc promet

Le runtime répond : `Error in class name, class not found: stxstring`. La classe est écrite, et aucun fichier d'entrée ne la charge !

Le contrat des couches dit que la couche base s'appuie sur le noyau, par héritage ou par délégation. Elle ne fait ni l'un ni l'autre pour les domaines communs : les deux partagent le moteur et implémentent deux fois ces domaines. La couche noyau elle-même est en développement face au moteur aujourd'hui. Ses classes de chaîne et de caractère appellent des noms du moteur qu'aucun fichier du moteur n'enregistre, comme le rapporte mon étude du dossier. Le bloc de la section 3 ne montre que des listes. Un chargeur qui amène un domaine de la base sans les autres n'est pas construit. Un tribunal qui lit le contrat des couches contre l'arbre et échoue en cas de dérive est une spécification, proposée dans cette étude. Ce qui est construit, ce sont deux implémentations qui partagent un moteur ; la ligne du contrat selon laquelle la base s'appuie sur le noyau est conçue.

Ce qui est absent : un programme entier et son moteur emballés dans un seul fichier sans aucune dépendance, et une plateforme cliente qui tourne sur le moteur.

8. SIX PORTES

Chacune des six portes lit cette image depuis l'endroit où elle se tient.

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous choisissez votre profondeur par le fichier que vous chargez, et un verbe veut dire le même acte sur une chaîne et sur une liste.
L'analyste	L'Atlas vous dit, voie par voie, où la plateforme est forte et où elle émerge. Construisez sur les voies fortes.
Le concepteur	Votre interface a un tribunal à elle, les verbes et les règles de la constitution, dans la même colonne que les gardes du code.
Le décideur	Posez deux questions à chaque étage : qu'y a-t-il dessous, et qui le juge. La réponse du moteur à la première est une liste écrite de bibliothèques épinglées.
Le formateur	L'image est la carte du livre. Chaque repensée des parties II à IV est une boîte de cette image.
L'agent	L'image existe comme des données que vous pouvez lire, l'Atlas comme un fichier, et chaque étage où vous écrivez a un tribunal qui vous lira.

9. EXERCICE

Dessinez votre propre solution comme ce chapitre dessine Softanza : des étages, et un tribunal à côté de chacun. Pour chaque boîte, écrivez le fichier qui la déclare et la garde qui la juge, et mettez dessus un mot d'étape : construit, conçu ou absent.

Puis déclarez-en deux parties avec `stzPlatformProfile`, donnez le profil à un `stzPlatform`, et avant d'exécuter quoi que ce soit, écrivez la ligne que vous attendez quand vous lui demandez de déployer avant de construire. Exécutez-le. Une boîte sans tribunal à côté est votre première trouvaille.

Je vous laisse celui-ci. La section 7 dit que les classes de chaîne de la couche noyau sont encore en développement face au moteur, et le bloc de la section 3 ne vous montre que des listes. Prenez le bloc de la section 7 et changez un mot : `stxString` devient `stkString`.

HARO

```
try
        o3 = new stkString("one and one")
catch
```

```
        ? cCatchError
done
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le, et lisez ce que répond le runtime. Puis retournez à la section 7 et trouvez quel nom est appelé, et qui aurait dû l'enregistrer. Quand vous l'aurez résolu, vous saurez pourquoi la ligne du contrat des couches selon laquelle la base s'appuie sur le noyau est dite conçue, et non construite.

Le chapitre suivant quitte l'image pour le code écrit à l'intérieur d'elle, et énonce la première loi : le code adhère à la façon dont une personne pense, et le tribunal le juge quel que soit celui qui l'a écrit. Son principe tient en une phrase : ce que vous pensez est ce que vous écrivez.

OÙ SONT LES PREUVES

NARRATION	`stzApp-building-your-first-app-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
NARRATION	`stz-emulating-the-whole-solution-narration.md`
NARRATION	`stz-vs-qt-unicode-narration.md`
SUR LE SITE	architecture
SUR LE SITE	platform
SUR LE SITE	platforms
SUR LE SITE	areas
DÉCISIONS DE L'ÉTUDE	12.2, 12.10, 12.12, 12.13, 12.15, 12.23, 12.29

PARTIE II

La programmation, repensée

> *Le code adhère à la manière de penser d'une personne, et le tribunal le juge, quel que soit son auteur.*

5 · Loi : le code adhère à la façon dont une personne pense, et le tribunal le juge quel que soit celui qui l'a écrit
6 · Le modèle mental, et les métaphores
7 · Le style d'écriture, et les sept buts
8 · Les fonctions comme expressions linguistiques, et la portée
9 · Les motifs de pensée : le Softanzuter, Reaxis et les moteurs
10 · La programmation devient raffinement

PARTIE II · CHAPITRE 5 · COLONNE

Loi : le code adhère à la façon dont une personne pense, et le tribunal le juge quel que soit celui qui l'a écrit

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends les deux choses que le terrain appelle style de code et je pose la question qu'il ne pose pas : si une ligne suit l'ordre dans lequel celui qui l'a écrite a raisonné. J'exécute l'exemple fondateur dans la forme la plus proche qui s'exécute encore, puis le modèle mental sur une liste, les formes du verbe, le paramètre nommé, la scène et l'historique d'une chaîne. Une promesse se tient sous chaque question. Je dis ce que j'ai laissé aller, l'implémentation propre de la bibliothèque, et ce que j'ai refusé de laisser aller, le style du code écrit avec elle. Je montre le tribunal qui existe aujourd'hui, sur les propres classes de la bibliothèque, et celui qui n'est que conçu. La devise n'a pas changé depuis 2020 : ce que vous pensez est ce que vous écrivez.

CE QUE L'ÂGE DE L'IA CHANGE ICI

Pendant des années, la plupart du code écrit avec Softanza l'a été d'une seule main, et cette main était la mienne. Le style de la bibliothèque était ma façon d'écrire. Puis les machines sont arrivées, et aujourd'hui la plupart du code d'ici est écrit par des modèles d'IA, comme la conception du volet de l'intelligence dit que c'est le cas partout, en une phrase : la plupart de la programmation d'aujourd'hui est faite par des agents et des modèles de langage, pas par des humains.

Cela change ce qu'est un style. Un style qui vit dans les habitudes d'une seule personne meurt quand cette personne cesse d'écrire chaque ligne. Il doit devenir quelque chose à quoi l'on peut tenir une machine, et quelque chose qu'une machine peut vérifier, sinon il s'amenuise. Dans cette maison, il s'est amenuisé, et je vous montrerai où. Alors ce mouvement du livre, La programmation repensée, s'ouvre sur une loi qui a deux moitiés. La seconde existe à cause de l'âge : le code adhère à la façon dont une personne pense, et le tribunal le juge quel que soit celui qui l'a écrit. Par le tribunal, j'entends la machine qui juge un code selon les règles de la maison ; aujourd'hui elle ne juge que les propres classes de la bibliothèque, et la section 5 dit comment.

Il y a une dépendance à dire à voix haute. Ce livre suit l'ordre de sa couverture, du familier au nouveau : la programmation, puis les langages, puis l'intelligence. La logique propre du dossier va dans l'autre sens. Le premier acte est de déclarer un langage ; la programmation se passe à l'intérieur d'un langage déclaré ; l'intelligence est gouvernée par les deux. Les formes du vocabulaire sont déjà déclarées dans un prototype de Haro, comme le note l'entrée du 3 octobre 2026 de sa charte. L'usage que ce chapitre décrit, la scène, la chaîne, le paramètre nommé, la promesse sous chaque ligne, est destiné à être déclaré aussi. Lisez donc cette loi en sachant que la partie III la soutient.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Dans notre industrie, le « style de code » veut dire deux choses, et aucune ne concerne la façon dont une personne pense. La première est la mise en page et le nommage, imposés par des machines : le guide de style de Python depuis 2001, le formateur de Go depuis 2009, et les formateurs que tout projet moderne exécute avant un commit. Ils règlent les disputes sur les espaces et les majuscules, et ils y sont bons. La seconde est l'interface fluide : des méthodes chaînées de sorte qu'une ligne se lise presque comme une phrase.

L'âge du code généré a ajouté une troisième pratique : vérifier après. Un modèle écrit, les tests et les linters lisent ce qu'il a écrit, et le modèle réessaie s'ils se plaignent. La ligne elle-même est jugée sur sa forme et son résultat. Savoir si elle suit

l'ordre dans lequel celui qui l'a écrite raisonnait est une question pour laquelle je ne connais aucun outil.

2. LE PREMIER PRINCIPE

Ma devise pour cela n'a pas changé depuis 2020 : **ce que vous pensez est ce que vous écrivez.**

La section 3 montre la devise à l'œuvre.

3. LA FORME SOFTANZA

La diapositive fondatrice mettait toute l'idée sur une page. Sur la gauche, les quatre étapes de la pensée d'une personne : je prends le mot « softanza » ; je le mets en majuscules ; j'obtiens ses caractères ; je les joins avec une espace. Sur la droite, les mêmes quatre étapes en une ligne de code, comme si le texte de gauche avait été collé et exécuté.

Cette ligne exacte ne s'exécute plus. Son troisième maillon, une méthode qui transformait les caractères en une liste de chaînes en une seule étape, n'est pas dans la bibliothèque aujourd'hui. Il ne survit que dans des fichiers archivés et dans un échantillon de travail, sous la forme `CharsQRT`. Je vous le dis d'abord parce qu'un chapitre de loi qui cache son propre exemple fondateur ne mériterait pas votre confiance. Quelle est la forme la plus proche qui s'exécute, avec les quatre étapes toujours visibles, et qu'offre la bibliothèque comme verbe dédié ? Exécutez les deux :

HARO

```
? StzListOfStringsQ( StzStringQ("softanza").UppercaseQ().Chars() ).ConcatenatedUsing(" ")
#--> S O F T A N Z A
? StzStringQ("softanza").UppercaseQ().Spacified()
#--> S O F T A N Z A
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux formes impriment `S O F T A N Z A`. L'idée est intacte ; le maillon fluide a été perdu. Une demande part vers la bibliothèque pour le rétablir.

Le modèle mental est la même idée au niveau d'un problème. Vous dites le problème en mots simples. Vous choisissez l'objet qui tient les données. Puis vous demandez, dans l'ordre : contient-il ce qui m'importe, combien, où ? Ensuite vous agissez sur ces positions, ou vous appelez le verbe qui le fait pour vous. Prenez une liste qui contient des chaînes vides. En contient-elle, combien, et où ? Exécutez :

HARO

```
o1 = new stzList([ "A", "", "B", "", "", "C" ])
? o1.ContainsEmptyStrings()
#--> TRUE
? o1.CountEmptyStrings()
#--> 3
? @@( o1.FindEmptyStrings() )
#--> [ 2, 4, 5 ]
o1.ReplaceEmptyStrings(:With = "~")
? @@( o1.Content() )
#--> [ "A", "~", "B", "~", "~", "C" ]
```

✓ exécuté : a affiché ce que ce bloc promet

La liste répond `TRUE`, puis `3`, puis `[ 2, 4, 5 ]`, et après l'acte elle se lit `[ "A", "~", "B", "~", "~", "C" ]`. Contenir, compter, trouver, agir. Chaque ligne est une étape de pensée, et les noms sont les mots que vous emploieriez pour la dire.

La forme du verbe dit ce qui arrive à l'objet. Que fait chaque forme à l'objet ? Exécutez-les sur une même chaîne :

HARO

```
o1 = new stzString("HAxxRxO")
? o1.Removed("x")
#--> HARO
? o1.Content()
#--> HAxxRxO
o1.Remove("x")
? o1.Content()
#--> HARO
? Q("haxxrxo").RemoveQ("x").UppercaseQ().Spacified()
#--> H A R O
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

`Removed` a imprimé `HARO`, et l'objet se lisait encore `HAxxRxO` ! `Remove` a ensuite agi sur lui, et `Content()` a lu `HARO`. La chaîne avec Q s'est terminée sur `H A R O`. Par convention de la bibliothèque, le participe passé vous donne une copie changée et laisse l'objet tranquille. Le verbe simple agit sur l'objet. Le verbe avec Q agit et rend l'objet, pour que l'étape suivante puisse suivre.

Faut-il retenir quel argument vient en second ? Un paramètre peut être nommé, et le nom se lit comme de l'anglais :

HARO

```
? Q("I love Haro").Replaced("love", :With = "♥")
#--> I ♥ Haro
o2 = new stzList([ "tea", "rice", "fish" ])
o2.SwapItems(:AtPositions = 1, :And = 3)
? @@( o2.Content() )
#--> [ "fish", "rice", "tea" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le texte imprime `I ♥ Haro` et la liste `[ "fish", "rice", "tea" ]`. Les noms `:With`, `:AtPositions` et `:And` disent quel argument est lequel.

Comment plusieurs questions sur une même chose se lisent-elles comme une liste ? Elles sont posées à l'intérieur de sa scène, un bloc d'accolades autour de l'objet. L'objet est nommé une seule fois :

HARO

```
Q("haro php HARO ruby Haro") {
        ? NumberOfOccurrence(:Of = "haro")
        #--> 1
        ? NumberOfOccurrenceCS(:Of = "haro", :CS = FALSE)
        #--> 3
        ? @@( FindAllCS("haro", FALSE) )
        #--> [ 1, 10, 20 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

La scène répond `1`, puis `3`, puis `[ 1, 10, 20 ]`. Le même mot a compté `1`, puis `3` dès que `:CS = FALSE` a été dit ! L'objet a été nommé une seule fois, et les trois questions se lisent comme une liste.

Les étapes de la pensée peuvent-elles être relues ? Un objet élevé avec son historique garde le résultat de chaque étape de sa chaîne :

HARO

```
? @@( Qh(12500).AddQ(500).RetrieveQ(1500).DivideByQ(500).MultiplyByQ(2).History() )
#--> [ 12500, 13000, 11500, 23, 46 ]
```

✓ exécuté : a affiché ce que ce bloc promet

L'exécution a imprimé `[ 12500, 13000, 11500, 23, 46 ]`. Cinq valeurs, la valeur de départ comprise, et la chaîne se termine à `46` ! L'article qui a raconté cela le premier promettait quatre valeurs, sans la première, et une valeur finale courte d'une unité, 45. La garde derrière le bloc énonce la règle d'aujourd'hui, sur une chaîne de texte : la valeur initiale, puis le résultat de chaque étape. Le livre prend la réponse de l'exécution, pas celle de l'article.

Les mêmes pas peuvent s'écrire comme une scène, la forme que je tiens pour la base du style : l'objet est nommé une fois, et chaque ligne entre les accolades lui parle.

HARO

```
# A number object that keeps its history
QH(12500) {
        # Inside the braces, each line speaks to the object
        Add(500)
        Retrieve(1500)
        DivideBy(500)
        MultiplyBy(2)

        # Its history, as one expressive value
        ? @@( History() )
        #--> [ 12500, 13000, 11500, 23, 46 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

La chaîne le dit en une phrase, la scène le dit en pas. Le livre emploie les deux : la phrase quand la pensée est une, la scène quand elle est une procédure. La forme classique nomme l'objet à chaque ligne, et beaucoup de lecteurs pensent ainsi :

HARO

```
o1 = new stzNumber(12500)
KeepHistory()

o1.Add(500)
o1.Retrieve(1500)
o1.DivideBy(500)
o1.MultiplyBy(2)

? @@( o1.History() )
#--> [ 12500, 13000, 11500, 23, 46 ]
KeepHistoryOFF()
```

✓ exécuté : a affiché ce que ce bloc promet

Trois formes, une réponse. Une quatrième est la phrase presque naturelle, `Naturally("...")`, que le chapitre 13 fait tourner sur du texte ; sur un nombre elle répond `Unsupported object type` à ce commit, et c'est son étape suivante.

Un mot sur `@@()`. Sur l'ancien langage hôte, `?` imprime une liste un élément par ligne ; `@@()` demande sa forme expressive, `[ 12500, 13000, 11500, 23, 46 ]`, celle qu'une personne écrit. J'ai demandé que Haro imprime cette forme par défaut ; alors `@@()` quittera les exemples.

Remarquez la ligne sous chaque question. Ce n'est pas un commentaire. C'est une promesse, ce que la ligne doit imprimer. Dans cette maison, elle est la forme d'un exemple depuis le premier jour : une courte histoire, quelques questions, et les réponses écrites dessous. C'est la première moitié de la loi, rendue visible : le code suit l'ordre de la pensée, et chaque étape dit ce qu'elle attend.

Maintenant la seconde moitié. Une règle qu'une personne doit retenir est du folklore. La bibliothèque transforme ses règles de maison en validateurs, pour qu'une personne ou un agent d'IA puisse vérifier une classe avant qu'elle n'atterrisse. Les règles sont quatorze, et la section 5 les compte ; cinq d'entre elles lisent le graphe du code, la structure que la bibliothèque construit à partir du texte d'une classe. Ici

une classe enfreint la convention de la maison selon laquelle un verbe finissant en Q rend l'objet. Cette règle a besoin des instructions de retour, que le graphe ne modélise pas, et elle s'exécute donc comme un passage sur le texte. Que dit le tribunal de la classe ? Exécutez :

HARO

```
cCode = "class Basket" + nl + "def Add(x)" + nl + "     return x" + nl + "def AddQ(x)" + nl + "
return 42" + nl
? StzCodeIsClean(cCode)
#--> FALSE
? StzCheckCode(cCode)[1][:rule]
#--> q-returns-object
```

✓ exécuté : a affiché ce que ce bloc promet

Le tribunal répond `FALSE`, et nomme la règle : `q-returns-object`. Il n'a pas demandé qui avait écrit la classe ! C'est toute la seconde moitié. Et la promesse sous chaque question est lue elle aussi par une machine : l'exécuteur de la bibliothèque juge une promesse comme une affirmation sur ce que la ligne a imprimé. La plus ancienne forme d'exemple de la maison est aussi sa plus récente forme de test.

Les chapitres de cette partie prouvent la loi, chacun d'un côté. Le chapitre 6 raconte le modèle mental et les quatre métaphores qui remplacent les boucles : marcher, vérifier, produire, exécuter. Le chapitre 7 raconte le style d'écriture comme quatorze règles qui lui sont propres, et non les quatorze règles de la maison du tribunal que vous venez de voir, et les sept buts de conception comme sa moitié architecturale. Le chapitre 8 raconte les formes que peut prendre le nom d'une fonction, la forme profonde parmi elles, et la portée nommée dans le verbe. Le chapitre 9 raconte les motifs de pensée : la machine qui réagit à ce qu'elle reconnaît, la programmation réactive en mots simples, et la ligne de temps et la grille. Le chapitre 10 raconte comment la programmation devient raffinement.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai laissé aller l'implémentation de la bibliothèque. Le code que les modèles ont écrit à l'intérieur de la bibliothèque est accepté comme bon. Il ne ressemble pas au mien. Ses variables locales portent des soulignés des deux côtés, des centaines de fois par millier de lignes. Ce sont une protection. Le langage sous-jacent à Softanza laisse la variable locale d'une bibliothèque écraser la globale d'un appelant, et la convention l'empêche. Je n'ai pas demandé l'ancien style là, et je refuserais une campagne pour l'imposer.

J'ai refusé de laisser aller le style du code écrit avec Softanza. Chaque ligne qu'un programmeur, une personne ou un agent d'IA, écrit avec la bibliothèque doit prendre le style historique. Ce style, c'est la scène, la chaîne une étape par ligne, le paramètre nommé, la forme du verbe, et la promesse sous chaque question. C'est la ligne que j'ai tracée quand j'ai vu ce qui s'était passé.

Ce qui s'est passé est mesurable. Mon étude du dossier a compté la promesse sous chaque ligne, par millier de lignes. Dans les courtes recettes que la maison garde, il y en a près de cent. Dans mes articles, soixante-quinze. Dans les tests classiques, cinquante-deux. Dans les gardes que les sessions ont écrites, moins d'une. Elles affirment sous une autre forme, un appel qui nomme l'attente. Elles l'ont adoptée parce que les comparaisons nues des anciennes suites imprimaient 0 ou 1, ce qui ne distingue pas un bloc qui a échoué d'un bloc qui n'a jamais tourné. Cet écart est comblé maintenant ; l'exécuteur lit la promesse comme une affirmation. L'étude qui les a comptées recommande une forme unique pour les nouveaux blocs de test : la forme classique, un titre, les questions et les promesses dessous. Les blocs convertis restent tels qu'ils sont. Le dossier contient cette recommandation, et aucune décision de moi à ce sujet.

J'ai refusé de réécrire le code quand le langage a changé de nom. Quand le langage sous-jacent est devenu Haro, le code de mes articles a été renommé et rien d'autre. Les formes sont l'atout ; le nom est une étiquette.

Les formes du vocabulaire sont devenues des données, et trois anciennes fonctionnalités ont été refusées comme grammaire. En Haro, les formes que peut prendre un verbe sont déclarées une fois et générées. Dans le prototype du langage, les noms livrés de la bibliothèque sont jugés contre cette déclaration. L'entrée du 3 octobre 2026 de sa charte rapporte 584 des 595 orthographes livrées de la classe de chaînes en accord. La conception de ce travail refuse trois choses que le dossier comptait comme des fonctionnalités : des noms qui tolèrent une faute d'orthographe, des paramètres dans n'importe quel ordre, et des noms de méthodes dans d'autres langues humaines. Le diaporama fondateur comptait les deux dernières ; la narration des formes comptait la première. La tolérance aux fautes et l'ordre libre restent dans la bibliothèque, en partie ; les noms multilingues ne survivent que dans un fichier archivé. Aucune des trois ne fait partie de la grammaire à laquelle une machine est tenue. Le refus est écrit dans le fichier des principes du langage, comme son septième principe, marqué conçu. Le livre l'imprime comme celui de la conception, et le dossier ne dit pas qu'il est le mien.

J'ai choisi les buts avant les lois. La conception de 2022 tenait sept buts, l'expressivité, la flexibilité, la fiabilité, la cohérence, les métaphores humaines, les abstractions pratiques et la maniabilité, et chaque fonctionnalité en servait un. Les règles de fonctionnement d'aujourd'hui sont des lois payées par des incidents, qui disent comment et non pourquoi. Le chapitre 7 remet les buts à côté des lois.

5. LE TRIBUNAL

`coderule_narrated` prouve qu'une règle de la maison est une règle sur le graphe du code et que les constats sortent sur du code planté. Elle prouve que le code propre passe, et que les règles de la maison de la bibliothèque sont quatorze. `testoor_promises_narrated` prouve que la promesse est jugée par une machine dans la forme propre à la maison. Chaque règle est plantée dans les deux sens : le mensonge qu'elle doit attraper, et la vérité qu'elle ne doit pas accuser. `05_containsemptystrings` est les étapes du modèle mental sur la même liste, un

bloc classique gardé comme une garde. `466_keeping_the_history_of_updates_of_a_softanza_objec` prouve l'historique d'une chaîne.

La section 6 met le tribunal à côté du terrain.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`coderule_narrated` 77 assertions
base/test/meta/coderule_narrated.ring

`testoor_promises_narrated` 46 assertions
base/test/testoor/testoor_promises_narrated.ring

`05_containsemptystrings` 5 assertions
base/test/list/05_containsemptystrings.ring

`466_keeping_the_history_of_updates_of_a_softanza_objec`
2 assertions
base/test/string/466_keeping_the_history_of_updates_of_a_softanza_objec.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène sur l'application des règles. La plupart des langages courants ont un formateur et un linter, et la plupart des projets les exécutent avant un commit ou en intégration continue ; c'est une connaissance générale, non une étude du dossier. Le tribunal qui jugerait le style d'un programme écrit avec Softanza est une spécification ; celui qui existe juge les propres classes de la bibliothèque.

Le terrain mène en portée et en gens. Il y a un marché du travail pour ses outils et aucun encore pour celui-ci, et les générateurs du terrain sont les plus grands qui existent.

Le terrain mène aussi, aujourd'hui, sur les propres pages de Softanza. L'étude qui a mesuré le dossier a trouvé que les blocs de code du site ne portent ni paramètre nommé ni scène : un dialecte réduit de la bibliothèque, montré à chaque visiteur. Un agent d'IA qui apprendrait du site apprendrait ce dialecte. Je n'ai pas mesuré de nouveau les blocs du site. Un style qui n'est pas montré n'est pas appris.

Là où je pense que Softanza se distingue, c'est dans ce à quoi le style est lié. Un formateur juge la forme d'une ligne ; une interface fluide rend une ligne agréable. Ici la ligne est liée à une étape de pensée, et la forme du verbe est un contrat sur l'objet. La promesse sous la ligne est l'exemple, le test et la documentation à la fois.

7. LA LIMITE

CONSTRUIT	les formes du vocabulaire dans des milliers de noms de méthodes ; les règles de la maison comme un tribunal sur le propre code de la bibliothèque ; la promesse lue par une machine dans la forme propre à la maison
CONÇU	un tribunal du style pour le code qu'un programmeur ou un agent d'IA écrit ; la grammaire d'écriture déclarée en Haro, pour qu'un modèle d'IA ne puisse écrire que dans elle
ABSENT	toute machine qui juge aujourd'hui le style d'un programme écrit avec la bibliothèque

Ce qui est construit : les formes du vocabulaire, dans des milliers de noms de méthodes ; les règles de la maison comme un tribunal sur le propre code de la bibliothèque ; la promesse jugée par une machine.

Ce qui est conçu : un tribunal du style pour le code qu'un programmeur ou un agent d'IA écrit avec la bibliothèque. Il lirait un argument positionnel là où un argument nommé existe, une question sans promesse, une chaîne sur une ligne au-delà de deux étapes. Et la grammaire d'écriture déclarée en Haro, pour que la sortie d'un modèle d'IA y soit tenue avant d'être écrite.

Ce qui est absent : toute machine qui juge, aujourd'hui, le style d'un programme écrit avec la bibliothèque. La chaîne de la diapositive fondatrice ne s'exécute pas telle qu'écrite. Dans ce livre, la discipline porte le nom de sa devise : ce que vous pensez est ce que vous écrivez.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vos lignes suivent votre raisonnement : sélectionner, contenir, compter, trouver, agir. Chaque question porte sa promesse, et un tribunal la lit que vous l'ayez écrite ou qu'un agent d'IA l'ait fait.
L'analyste	La méthode part de votre problème en mots simples. Les mots-clés que vous employez deviennent les noms des étapes.
Le concepteur	La forme du verbe est un contrat d'interface : par convention de la bibliothèque, le participe passé ne touche pas l'objet. Un tribunal pour cette règle est conçu, non construit.
Le décideur	Je tiens cela pour une vue, non pour un résultat : un style auquel une machine peut être tenue peut être relu but par but, non ligne par ligne. C'est la seule lecture qu'une personne puisse donner à des centaines de milliers de lignes.
Le formateur	Enseignez la discipline entière, et dans l'ordre : formuler, puis exprimer, puis être jugé.
L'agent	Vous serez corrigé par la bibliothèque, non par le folklore. Écrivez la question, puis la promesse dessous.

9. EXERCICE

Prenez la liste `[ "tea", "", "rice", "", "fish", "" ]`. Écrivez dans l'ordre les questions du modèle mental, une ligne chacune : contient-elle des chaînes vides, combien, où. Puis un acte, avec un paramètre nommé. Sous chaque question, écrivez la ligne qu'elle doit imprimer, avant d'exécuter quoi que ce soit.

Exécutez. Puis demandez à un agent d'IA de résoudre le même problème, et jugez sa réponse avec deux questions seulement : chaque ligne suit-elle une étape de pensée, et chaque question porte-t-elle sa promesse.

Je vous laisse celui-ci, exprès. La classe Basket de la section 3 échoue devant le tribunal parce que `AddQ` renvoie `42`. Changez cette seule valeur en `This` et exécutez de nouveau la question :

HARO

```
cCode = "class Basket" + nl + "def Add(x)" + nl + "     return x" + nl + "def AddQ(x)" + nl + "
return This" + nl
? StzCodeIsClean(cCode)
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Ne vous arrêtez pas à la réponse du tribunal. Demandez-vous si la classe, telle qu'elle est, suit la façon dont une personne pense, et ce que le tribunal a lu pour vous répondre. Je ne vous donne pas le verdict : la section 6 dit quel tribunal existe, et la section 7 dit ce qu'il ne lit pas. Quand vous aurez tranché, vous saurez pourquoi un tribunal qui laisse passer une classe n'a pas encore jugé le style d'un programme écrit avec la bibliothèque.

Le chapitre suivant est le modèle mental, et les métaphores. Là où vous écririez une boucle, il vous demande de nommer plutôt le mouvement que vous faites : marcher, vérifier, produire ou exécuter.

OÙ SONT LES PREUVES

NARRATION	`stz-mental-mode-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
NARRATION	`stz-functions-as-linguistic-expressions.md`
NARRATION	`stzobject-history-narration.md`
NARRATION	`stz-near-natural-language-programming.md`
SUR LE SITE	learn
SUR LE SITE	craft
SUR LE SITE	principles
DÉCISIONS DE L'ÉTUDE	12.2, 12.3, 12.14, 12.16, 12.17, 12.21

PARTIE II · CHAPITRE 6

Le modèle mental, et les métaphores

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 7 bloc(s) sur 7 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends les sept étapes du modèle mental, dont le chapitre 5 a exécuté les quatre du milieu sur une liste, et je les exécute sur une chaîne, dans le même ordre. Puis j'en viens à la boucle, l'endroit où les étapes se répètent, et aux quatre métaphores que la bibliothèque vous donne à la place : le marcheur, le vérificateur, le producteur et l'exécutant. Je montre la condition que les quatre portent à l'intérieur d'un argument, la condition W que le moteur lit comme des données, et son jumeau WF, qui exécute du code hôte. Je dis pourquoi j'ai fait des métaphores un but de conception, pourquoi j'ai remplacé `eval` par un langage que le moteur possède, et où le code des articles sur lesquels ce chapitre repose n'est pas encore exécutable à ce commit. Le principe sous tout cela est celui du chapitre : quand vous écririez une boucle, nommez le mouvement que vous faites, marcher, vérifier, produire ou exécuter.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a un seul outil pour répéter une chose, la boucle, et une famille de cousins qui la cachent : l'itérateur, la compréhension, le flux avec son map, filter et reduce, la lambda qui porte une condition. Ce sont de bons outils, et ils partagent tous une habitude. Ils sont écrits dans le langage du mouvement de la machine : un index qui commence à zéro, un compteur qu'on incrémente, une variable qui accumule. Une personne qui lit la boucle doit retrouver la question qu'elle pose.

Le terrain a aussi un vocabulaire de métaphores pour la programmation, mais il les garde dans la documentation. Le code reste une boucle.

2. LE PREMIER PRINCIPE

Dites le problème en mots simples, posez les mêmes questions dans le même ordre à n'importe quel objet, et quand vous écririez une boucle, nommez le mouvement que vous faites : marcher, vérifier, produire ou exécuter. La page Craft du site le dit, telle que je l'ai lue le 9 octobre 2026 : « Une boucle cache quel mouvement elle fait. Softanza nomme les quatre mouvements, et une tâche est décomposée en eux à la place. » Cette phrase est celle du site, non la mienne ; la phrase ci-dessus est le principe de ce chapitre.

3. LA FORME SOFTANZA

Le modèle mental compte sept étapes, et ce sont celles dans l'ordre desquelles une personne demande. Définir le problème en termes naturels et en prendre les mots-clés. Choisir l'objet qui tient les données. Demander s'il contient ce qui vous importe. Demander combien. Demander où. Agir sur ces positions, ou appeler le verbe qui le fait. Et la septième : célébrer, et recommencer. Au chapitre 5, vous avez exécuté ces étapes sur une liste de chaînes vides. Elles n'appartiennent pas à la liste. Tiennent-elles sur une chaîne ? Les voici, dans le même ordre :

HARO

```
Q("You and you") {
        ? ContainsCS("YOU", FALSE)
        #--> TRUE
        ? NumberOfOccurrenceCS("YOU", FALSE)
        #--> 2
        ? FindNthCS(2, "YOU", FALSE)
        #--> 9
        ReplaceSection(9, 11, :With = "Me")
        ? Content()
        #--> You and Me
}
```

✓ exécuté : a affiché ce que ce bloc promet

Contenir imprime `TRUE`, compter imprime `2`, trouver imprime `9`, et agir laisse `You and Me`, avec le même suffixe qui dit si la casse compte. Les trois questions que vous posez à une chaîne sont les trois que vous avez posées à une liste.

Quand les étapes se répètent, la boucle apparaît, et c'est là que la bibliothèque vous donne quatre métaphores à la place. Chacune nomme un mouvement.

Le Marcheur se déplace le long d'une route. Un marcheur est une carte de tout son voyage, construite à sa création. Il garde les positions qu'il visitera, les positions qu'il sautera, où il se tient et ce qui reste. Que sait-il avant de faire un pas ? Demandez-le-lui :

HARO

```
Wk(1, 10, 2) {
        ? @@( Walkables() )
        #--> [ 1, 3, 5, 7, 9 ]
        ? @@( Unwalkables() )
        #--> [ 2, 4, 6, 8, 10 ]
        ? @@( Walk() )
        #--> [ 1, 3 ]
        ? CurrentPosition()
        #--> 3
        ? @@( RemainingWalkables() )
        #--> [ 5, 7, 9 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

`Walk()` est un pas, et il imprime le chemin du mouvement, de là où il se tenait à là où il a atterri : `[ 1, 3 ]`. Une boucle ne garde pas son itinéraire comme une valeur qu'on peut interroger. Un marcheur le fait, avant de bouger. Que fait-il au bord de la route ? Au-delà de la fin, quand la marche est en cours, il s'arrête en silence. Avant le début, il refuse :

HARO

```
oWalker = Wk(1, 10, 3)
? @@( oWalker.WalkNSteps(10) )
#--> [ 1, 4, 7, 10 ]
? @@( oWalker.Walk() )
#--> [ 10 ]
oWalker = Wk(1, 10, 3)
```

```
try
        oWalker.WalkBackward()
catch
        ? cCatchError
done
#--> Can't walk backward! Already at the beginning of walkable positions.
```

✓ exécuté : a affiché ce que ce bloc promet

Demandez dix pas à une marche qui en compte quatre et vous obtenez les quatre : `[ 1, 4, 7, 10 ]`. Marchez encore et il imprime `[ 10 ]`, la position où il se tient, sans erreur. Demandez-lui de marcher en arrière depuis le début et il refuse ! Ses propres mots : `Can't walk backward! Already at the beginning of walkable positions.` L'article soutient que cela reflète notre façon de marcher dans le monde physique. On s'arrête quand on heurte un mur derrière soi, et c'est une erreur. Quand la route devant finit, on s'arrête en silence.

Le Vérificateur demande oui ou non. Tous, n'importe lequel, ou où, avec la condition écrite comme une chaîne, la condition W, W pour where :

HARO

```
Q([ "A", 5, "B", "C" ]) {
        ? CheckW('{ isString(@item) and IsUppercase(@item) }')
        #--> FALSE
        ? ContainsW('{ isNumber(@item) }')
        #--> TRUE
        ? @@( FindW('{ isNumber(@item) }') )
        #--> [ 2 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Un élément échoue au test, donc `CheckW` imprime `FALSE` ; un nombre est dans la liste, donc `ContainsW` imprime `TRUE` ; et `FindW` dit où, `[ 2 ]`.

Le Producteur produit des données neuves à partir des anciennes, et laisse les anciennes tranquilles. L'Exécutant est son pendant : il change les éléments là où la condition tient. Lequel des deux touche la liste ? Mettez-les côte à côte sur la même liste et regardez :

HARO

```
o1 = new stzYielder([ 1, -2, 3, -4, 5 ])
o1 {
        ? @@( Filter(:IsPositive) )
        #--> [ 1, 3, 5 ]
        ? @@( Map(:Abs) )
        #--> [ 1, 2, 3, 4, 5 ]
        ? Reduce(:Sum)
        #--> 3
}
Q([ 1, "haro", 3, "python", 5, "ruby" ]) {
        ? @@( YieldW('isString(@item)', 'upper(@item)') )
        #--> [ "HARO", "PYTHON", "RUBY" ]
        ? @@( Content() )
        #--> [ 1, "haro", 3, "python", 5, "ruby" ]
        PerformW('isString(@item)', 'upper(@item)')
        ? @@( Content() )
        #--> [ 1, "HARO", 3, "PYTHON", 5, "RUBY" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Regardez les deux sorties de `Content()`, avant et après l'appel à `PerformW`. Le producteur a rendu `[ "HARO", "PYTHON", "RUBY" ]`. La liste n'a pas été touchée ! L'exécutant a changé la liste : `[ 1, "HARO", 3, "PYTHON", 5, "RUBY" ]`. Le verbe vous dit lequel, si bien qu'une conception peut dire « cette étape ne fait que lire » et le penser.

Maintenant ce que les quatre partagent : une condition voyage à l'intérieur d'un argument. C'est une petite chaîne, gardée dans une variable si vous voulez. Elle se place au même endroit pour find, pour items et pour contains. Et elle a un jumeau qui prend une vraie fonction :

HARO

```
cCond = '{ @item > 5 }'
Q([ 3, 8, 5, 12, 1, 9 ]) {
        ? @@( FindW(cCond) )
        #--> [ 2, 4, 6 ]
        ? @@( ItemsW(cCond) )
        #--> [ 8, 12, 9 ]
        ? ContainsW(cCond)
        #--> TRUE
        ? @@( FindWF( func x { return x > 5 } ) )
```

```
        #--> [ 2, 4, 6 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux s'accordent : `FindW` avec la chaîne et `FindWF` avec la fonction impriment tous deux `[ 2, 4, 6 ]`. `ItemsW` rend les éléments eux-mêmes, `[ 8, 12, 9 ]`.

La forme en chaîne est la condition W ; la forme en fonction est WF. Elles se ressemblent, et la différence compte. Le moteur lit une condition W comme des données, dans un petit langage à lui : des littéraux, `@item`, l'arithmétique, les comparaisons, et une courte liste de fonctions. Il ne peut pas appeler votre code. Alors que devient une condition qui essaie ? Elle n'est pas exécutée :

HARO

```
o1 = new stzList([ 3, 8, 5, 12, 1, 9 ])
? @@( o1.FindW('{ @item > 5 and print("hacked") }') )
#--> [ ]
? @@( W('{ @item > 5 }') )
#--> [ "where", "{ @item > 5 }" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Rien n'a été imprimé ! Les positions sont revenues vides, `[ ]`, et `W()` montre la condition pour ce qu'elle est, des données : `[ "where", "{ @item > 5 }" ]`. Une condition W peut être stockée, montrée à une personne et approuvée avant de s'exécuter, parce qu'elle ne peut pas s'échapper. Le jumeau WF exécute du code hôte, et la table propre de la bibliothèque dit qu'il n'est pas sûr avec une entrée à laquelle vous ne faites pas confiance.

Le code de l'article « Bridging Minds and Code » ne s'exécute pas à ce commit, comme le dit la section 7.

Les trois modèles de cet article, le sémantique, le mental et celui de l'expression, sont une seule idée en trois endroits. Le modèle sémantique est le monde du programme comme connaissance : une pomme est un fruit. Le modèle mental est ce que vous venez de faire : créer un objet, y stocker des données, y trouver des choses,

agir sur les positions. Le modèle d'expression est le code qui le dit : choisir l'objet, taper le nom de l'action, continuer le flux d'actions, convertir le résultat en un autre objet. Continuer ainsi jusqu'à la réponse finale. Alors je les raconte en mots.

La section suivante donne la décision, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai fait des métaphores un but de conception. La présentation de 2022 liste les « métaphores centrées sur l'humain » comme l'un des sept buts de conception, et ses quatre fonctionnalités sont ces quatre. Un but est ce qui rend une fonctionnalité justifiable ou refusable. Je tiens cela pour une vue, non pour un résultat. Alors je n'ai pas laissé les métaphores comme une politesse de la documentation. Elles sont l'une des raisons pour lesquelles la bibliothèque a la forme qu'elle a.

J'ai remplacé `eval` par un langage que le moteur possède. L'emplacement de la condition était compilé en évaluant la chaîne comme du code, avec une forme étendue plus expressive mais plus lente à côté. J'ai retiré les deux. Le moteur lit maintenant la condition dans son propre petit langage, et la bibliothèque garde la forme en code hôte comme un verbe séparé, WF, avec son avertissement. J'ai refusé le code arbitraire venu d'un texte, et j'ai refusé de séparer la vitesse de l'expressivité : la forme étendue est retirée parce que la simple fait tout ce qu'elle faisait.

J'ai gardé les sept étapes telles qu'écrites. Quand le cours a été conçu, une lecture du noyau technique a trouvé cinq étapes : select, contains, count, find, apply. Le cours enseigne les sept comme la narration source les a. Personne ne renumérote la source.

J'ai laissé la règle du bord être un principe de conception. Avant le début est une erreur, et au-delà de la fin est ignoré. La narration donne sa raison en une phrase : cela reflète notre façon de marcher. Le dossier ne la date pas comme une décision, alors je la raconte comme la conception de la bibliothèque.

Le tribunal vient ensuite, et ses gardes ne couvrent pas toutes les lignes ci-dessus.

5. LE TRIBUNAL

Les quatre métaphores n'ont pas de garde unique. Le producteur en a une narrée, `00_yielder_narrated` : quatre opérations map, trois opérations filter et une somme. Le marcheur a `597_walker_basics`, des marcheurs nommés et ce qu'ils ont parcouru, et `495_walking_until_and_until_before`, une marche qui s'arrête sur une condition, y compris l'élément qui l'arrête, ou non ; toutes deux testent les marcheurs nommés et les marches de la liste, par `StzListQ`, non le marcheur `Wk` des blocs ci-dessus. Le vérificateur a `611_checkw`, qui dit qu'une vérification est fausse quand un seul élément échoue. L'exécutant a `621_perform`, qui dit qu'il change la liste en place. `483_w_and_wf_forms_in_conditional_code` prouve qu'une condition W écrite avec des mots de curseur comme `@NextNumber` et la même écrite avec de l'arithmétique d'index brute comme `This[@i+1]` donnent les mêmes positions.

Aucune des gardes nommées ici ne couvre `Walkables`, `WalkNSteps` ni `WalkBackward` ; ces lignes reposent sur les exécutions gardées pour ce chapitre.

Ensuite, ce que le terrain fait mieux que le marcheur.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`00_yielder_narrated` 8 assertions
base/test/yielder/00_yielder_narrated.ring

`597_walker_basics` 6 assertions base/test/list/597_walker_basics.ring

`611_checkw` 1 assertions base/test/list/611_checkw.ring

`621_perform` 2 assertions base/test/list/621_perform.ring

`483_w_and_wf_forms_in_conditional_code` 2 assertions
base/test/list/483_w_and_wf_forms_in_conditional_code.ring

`495_walking_until_and_until_before` 4 assertions
base/test/list/495_walking_until_and_until_before.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en paresse. Un générateur produit l'élément suivant à la demande et peut être sans fin. Le marcheur construit une carte de tout son voyage à l'instant où il est fait. C'est un gain pour la prévoyance : vous pouvez demander quelles positions il sautera avant qu'il bouge. C'est une perte pour un grand parcours, ou un parcours sans fin.

Le terrain mène en ouverture. Une lambda porte n'importe quelle logique. Le langage W est borné exprès, et le prix de la borne est le second chemin, WF, qui a toute la puissance de l'hôte et il est fait pour une entrée à laquelle on fait confiance. Il y a aussi un coût plus discret que j'ai mesuré une fois, sur un cas : une condition qui appelle quelque chose que le langage ne connaît pas donne une liste vide et aucune erreur.

Le terrain mène en portée et en outillage. Les itérateurs et les compréhensions sont dans tous les langages courants, avec le soutien des éditeurs, la vérification des types et une littérature. Ces quatre métaphores existent dans une seule bibliothèque.

Et je n'imprimerai pas la comparaison que fait la bibliothèque elle-même des marcheurs avec les boucles et les itérateurs. Sa table a treize lignes, toutes en faveur du marcheur, aucune exécutée. Une table vaut d'être imprimée quand ses lignes ont été exécutées et qu'une perte figure à côté des gains ; pour celle-ci, c'est l'étape suivante.

Ensuite, la limite, dans les mots d'étape construit, conçu et absent.

7. LA LIMITE

CONSTRUIT	le marcheur à une dimension, le vérificateur, le producteur et l'exécutant sur les listes ; le langage de condition compilé par le moteur, avec son jumeau en code hôte
CONÇU	les quatre métaphores comme modules natifs du moteur (seul le producteur en est un) ; un marcheur sur toute structure indexée
ABSENT	du code exécutable pour les trois modèles de l'article « Bridging Minds and Code » ; la table des bords de la narration du marcheur telle qu'écrite

Ce qui est construit : le marcheur à une dimension, dans la couche max, et les marches sur les listes ; le vérificateur comme `CheckW`, `ContainsW` et `FindW` ; le producteur comme une classe adossée au moteur, et comme `YieldW` ; l'exécutant comme `PerformW` ; et le langage de condition W compilé par le moteur.

Ce qui est conçu : les métaphores comme modules natifs du moteur. La conception du moteur marque les quatre comme prévus ; seul le producteur a un fichier de moteur. Et un marcheur qui ne porte plus du tout sur les listes mais sur toute structure indexée.

Six choses sont absentes, ou restent à aligner sur leurs articles, à ce commit, d'après mes exécutions.

- La table de l'article sur la marche au bord et la bibliothèque diffèrent dans deux cas ; les aligner est l'étape suivante. Un pas unique plus grand que l'intervalle lève

une erreur là où la table promet le silence, et un chemin de directions mêlées lève une erreur là où elle promet l'acceptation. Les deux principes tiennent dans les cas montrés plus haut. Le livre énonce le principe et n'imprime que les cas qui s'exécutent ; la table n'est pas imprimée.

- Le code de l'article « Beyond Loops » diffère par endroits. L'historique d'un marcheur est une liste de marches ; une marche en arrière ne prend plus d'argument ; la direction s'imprime « backward » là où l'article dit « reverse ». L'exécution de l'article par le site a gardé dix blocs sur dix-huit.
- Les trois modèles de « Bridging Minds and Code » : aucun de ses codes ne s'exécute tel qu'écrit ; dans l'exécution du site, douze blocs ont levé une erreur et un a différé. Alors je les ai racontés en mots.
- Le getter `EmptyStrings()` de l'article du modèle mental est absent.
- Sur la classe dédiée du vérificateur, `ContainsW` est en développement : il lève une erreur aujourd'hui. La marche en arrière avec condition du marcheur de chaînes est absente.
- Et « pas d'eval » est encore en route vers le reste. Le résolveur quasi naturel, l'autre direction du chapitre 13, évalue encore du code hôte. Il en va de même de `StringsW` et de `ItemsWhere` sur la liste, et de `ContainsW` sur la classe dédiée du vérificateur : chacun appelle encore `eval` sur une chaîne de condition. Ce que j'ai dit plus haut, que le moteur lit une condition W et ne peut pas l'exécuter comme du code, vaut pour le chemin du moteur ; pour ces trois, c'est l'étape suivante.

Viennent ensuite les six portes, du programmeur à l'agent.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Posez les quatre mêmes questions à n'importe quel objet, et quand vous tendez la main vers une boucle, nommez le mouvement : marcher, vérifier, produire ou exécuter.
L'analyste	Dites le problème en mots simples, et les verbes que vous avez employés sont les étapes : « sont-ils tous », « lesquels », « donne-moi les grands », « rends-les négatifs ».
Le concepteur	Le producteur ne change jamais la liste et l'exécutant la change, et le verbe vous dit lequel.
Le décideur	Une condition W est une chaîne que le moteur lit et ne peut pas exécuter comme du code, si bien qu'elle peut être stockée, montrée et approuvée. Le prix est un petit langage, et un appel qu'il ne connaît pas donne une liste vide, non une erreur.
Le formateur	Le livre d'autoformation a déjà l'ordre : trouver, puis appliquer, puis marcher, demander, produire et agir.
L'agent	Écrivez la condition comme la chaîne, non comme du code. Quand la réponse est vide, vérifiez les mots de la condition avant de croire au vide.

9. EXERCICE

Prenez la liste `[ 4, 9, 12, 7, 20, 3 ]`. Quatre phrases sont données. Une : sont-ils tous au-dessus de 2 ? Deux : donnez-moi, comme une nouvelle liste, ceux au-dessus de 8, doublés. Trois : rendez négatifs ceux au-dessus de 8, dans la liste elle-même. Quatre : quelles positions sont 2, 4 et 6 ?

Pour chacune, dites d'abord quel mouvement c'est, marcher, vérifier, produire ou exécuter, et écrivez sous la ligne que vous allez écrire ce que vous attendez qu'elle imprime. Puis écrivez la ligne, avec la condition comme une chaîne partout où il y a une condition, et exécutez-la. Après la deuxième et la troisième, imprimez de nouveau la liste. Dites, avant de regarder, dans laquelle des deux la liste a changé.

Je vous laisse celui-ci, exprès. Voici le bloc de la section 3 avec un seul changement : la liste est celle de l'exercice, et je n'imprime rien dessous.

HARO

```
o1 = new stzList([ 4, 9, 12, 7, 20, 3 ])
? @@( o1.FindW('{ @item > 5 and print("hacked") }') )
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. La condition commence par demander si un élément est au-dessus de 5. Regardez ce qui revient, et ce qui ne revient pas, et dites pourquoi avant de retourner aux sections. La section 3 dit ce que le moteur fait d'une condition qui appelle du code ; la section 6 dit ce que cela coûte, et la section 8 dit ce qu'un agent d'IA doit en faire. Quand vous l'aurez résolu, vous saurez pourquoi une réponse vide n'est pas encore la preuve que rien n'a correspondu.

Le chapitre suivant passe au style d'écriture, et aux sept buts. Son principe tient en une phrase : ce que vous pensez est ce que vous écrivez.

OÙ SONT LES PREUVES

NARRATION	stz-mental-mode-narration.md
NARRATION	stzwalker-beyond-loops-the -walker-metaphor.md
NARRATION	stzwalker-walking-at-the-edge-narration.md
NARRATION	stz-conditional-code-narration.md
NARRATION	stz-bridging-minds-and-code.md
SUR LE SITE	learn
SUR LE SITE	craft
DÉCISIONS DE L'ÉTUDE	12.3, 12.6, 12.14, 12.16, 12.17

PARTIE II · CHAPITRE 7

Le style d'écriture, et les sept buts

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends le reste du style d'écriture, comme une grammaire de quatorze règles. J'imprime l'état où chaque règle se trouve au commit sur lequel ce livre s'exécute, un bloc là où elle s'exécute et une limite là où elle ne s'exécute pas. Sous la grammaire je mets les sept buts, j'en montre trois dans des blocs et je laisse à l'étude les quatre que je n'ai pas exécutés. J'imprime le tribunal de la maison, les quatorze règles que la bibliothèque tient sur son propre code. Je dis ce qu'il ne lit pas : le code d'un programme écrit avec la bibliothèque. Le principe est celui du chapitre 5, et je ne l'argumente pas de nouveau : ce que vous pensez est ce que vous écrivez.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le mot *style* veut dire deux choses dans le terrain, et je dois dire laquelle j'entends.

Le premier est le style comme mise en page, imposé par un outil. Le guide de style de Python date de 2001, le formateur de Go de 2009, et tout projet exécute un linter avant un commit. Ils jugent la forme d'une ligne : espaces, majuscules, longueur, noms inutilisés. Ils y sont bons, et ils tournent par défaut. Ils ne disent rien de savoir si une ligne suit l'ordre dans lequel son auteur raisonnait.

Le second est le petit langage à l'intérieur du langage. Les programmeurs écrivent déjà de petits langages dans des chaînes, et personne n'appelle cela un style : une expression régulière, un texte SQL, une chaîne de format telle que `"%5.2f"`. Chacun est une notation séparée, avec son propre outil, apprise à part.

Pour les buts, le terrain dit *attributs de qualité*. La norme ISO de qualité logicielle, de 2011, en liste huit, et les méthodes d'architecture les classent avant qu'un système soit dessiné ; c'est une connaissance générale, non une étude du dossier. Les sept buts de ce chapitre sont une autre liste, choisie pour un autre usage. Ils ne disent pas ce qu'un produit promet à ses utilisateurs. Ils disent ce qu'une fonctionnalité doit servir pour être admise dans une seule bibliothèque.

2. LE PREMIER PRINCIPE

Ce que vous pensez est ce que vous écrivez. C'est la phrase du chapitre 5, tirée du schéma de 2020, et je ne l'argumente pas de nouveau. Le chapitre 5 a pris cinq règles du style. Ce chapitre prend le reste, comme une grammaire, puis les buts sous lesquels la grammaire a été conçue.

Pour les buts, le point est le mien, et non une citation datée. Une architecture logicielle orientée par les buts est la seule façon pour un architecte de maîtriser un projet de cette taille. Je tiens cela pour une vue, non pour un résultat. Ce qui la soutient sur cette page est un seul cas : la matrice de 2022 a servi de squelette à un architecte pour environ trois cent mille lignes écrites à la main (section 6).

3. LA FORME SOFTANZA

Mon étude du dossier a écrit le style en quatorze règles, à partir de la lecture du code. Ce sont des lectures du code, énoncées comme des règles, et non une norme que quelqu'un a signée. Les voici avec l'état où chacune se trouve au commit sur lequel ce livre s'exécute. Je mets la table d'abord parce qu'elle montre quelles règles portent un bloc et lesquelles portent une limite.

#	RÈGLE	ÉTAT À CE COMMIT
1	Une valeur est élevée à son objet par `Q()` ou un élévateur typé	s'exécute ; chapitre 5
2	Une scène est un bloc d'accolades sur un seul objet	s'exécute ; chapitre 5
3	Chaque ligne imprimée porte sa réponse sous `#-->`	jugé par une machine ; chapitre 5
4	Un paramètre est nommé, et le nom se lit comme de l'anglais	s'exécute ; chapitre 5, et ici dans n'importe quel ordre

#	RÈGLE	ÉTAT À CE COMMIT
5	Une chaîne reflète l'ordre de la pensée, une étape par ligne	s'exécute ; chapitre 5
6	La forme du verbe dit ce qui arrive	s'exécute ; chapitres 5 et 8
7	Les suffixes et les préfixes sont petits et veulent dire une seule chose chacun	en partie : `CS`, `XT`, `viz` s'exécutent ; les familles `FF`, `dft` et `inf` du diaporama ne sont pas définies
8	Une fonction a des alternatives pour la façon dont une personne la dit	en partie
9	Précision plutôt que généralité, quand cela ne coûte rien	en partie
10	Le code voyage à l'intérieur d'un argument comme une petite chaîne	l'emplacement s'exécute ; le `@` sur le nom d'un paramètre est retiré
11	De petits langages littéraux vivent dans des chaînes	s'exécute sous une forme ; la forme à accolades lève une erreur
12	Une phrase peut être quasi naturelle	pas de ce chapitre ; aucune des trois formes du diaporama ne s'est exécutée telle qu'écrite
13	Une fonction peut être nommée dans n'importe quelle langue humaine	absent ; section 7
14	Un commentaire parle avant le code, dans la voix du maître	s'exécute

La règle 11 d'abord, parce que c'est la plus étrange. Le langage hôte compte en entiers et coupe les décimales. Que garde un intervalle écrit à l'intérieur d'une chaîne ? Les commentaires ci-dessous parlent avant le code, dans la voix du maître, ce qui est la règle 14. Exécutez-le :

HARO

```
# The host's own range counts in integers: the decimals are cut. Softanza reads
# a small language inside a string, and keeps the decimals, the numbered words,
# and the letters of any script:
? @@( 1.02 : 3.08 )
#--> [ 1, 2, 3 ]
? @@( L("1.02:1.05") )
#--> [ 1.02, 1.03, 1.04, 1.05 ]
? @@( L('"♥1" : "♥3"') )
#--> [ "♥1", "♥2", "♥3" ]
? @@( L(' "ج" : "ا" ') )
#--> [ "ج" ,"ث" ,"ت" ,"ة" ,"ب" ,"ا" ]
```

✓ exécuté : a affiché ce que ce bloc promet

L'hôte répond `[ 1, 2, 3 ]` à `1.02 : 3.08` et coupe les décimales. La chaîne répond `[ 1.02, 1.03, 1.04, 1.05 ]`, et les deux bornes arabes donnent six lettres ! Softanza lit un petit langage d'intervalles à l'intérieur de la chaîne et garde les décimales, les mots numérotés et les lettres de n'importe quelle écriture. Le langage d'intervalles est un argument comme un autre. Donne-t-il les remplacements pour chaque occurrence, dans l'ordre, et la forme étendue reboucle-t-elle la palette quand les occurrences la dépassent en nombre ?

HARO

```
o1 = new stzString("haro php ruby haro python haro")
o1.ReplaceByMany("haro", L('"♥1" : "♥3"'))
? o1.Content()
#--> ♥1 php ruby ♥2 python ♥3
o2 = new stzString("haro php ruby haro python haro ruby haro")
o2.ReplaceByManyXT("haro", L('"♥1" : "♥3"'))
? o2.Content()
#--> ♥1 php ruby ♥2 python ♥3 ruby ♥1
```

✓ exécuté : a affiché ce que ce bloc promet

La forme étendue a rebouclé la palette, et ♥1 est revenu à la fin !

Une liste peut aussi être portée comme une chaîne dans sa forme courte, reconnue, convertie entre les formes et réévaluée :

HARO

```
? StzStringQ('1:3').IsListInString()
#--> TRUE
? @@( StzStringQ('[1,2,3]').ToListInShortForm() )
#--> "1 : 3"
? @@( StzStringQ(' "A":"C" ').ToList() )
#--> [ "A", "B", "C" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Règle 10 : le code voyage à l'intérieur d'un argument comme une petite chaîne avec un espace réservé, `@char` pour les caractères d'une chaîne et `@item` pour les éléments d'une liste. Le chapitre 6 a montré l'emplacement sur une liste. Ici il est sur une chaîne, où il trouve et retire par une règle et non par une boucle :

HARO

```
Q("SOooooFTAaaannnNZA") {
        ? @@( FindW('{ Q(@char).IsUppercase() }') )
        #--> [ 1, 2, 7, 8, 9, 16, 17, 18 ]
        RemoveW('{ Q(@char).IsLowercase() }')
        ? Content()
        #--> SOFTANZA
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le diaporama de 2022 écrit le même appel avec un emplacement nommé, `:Where = '...'`, et cette forme lève une erreur sur une chaîne à ce commit. Je ne la montre pas. Le même diaporama met un `@` à la fin du nom d'un paramètre, `:With@`, pour marquer que sa valeur est du code à évaluer. Cette forme est retirée. L'en-tête d'une garde le dit : les formes fondées sur eval sont retirées. Le diaporama lui-même marque la forme « TODO ». Le code voyage maintenant dans l'emplacement comme une chaîne bornée que le moteur évalue, ou comme une fonction.

Règle 8 : le même acte peut se dire en deux ordres de mots, avec le nom du conteneur, ou avec le côté épelé de quatre façons, gauche ou début, droite ou fin ; deux des quatre s'exécutent ci-dessous. Chaque paire est un seul acte, avec deux définitions. La propre narration de la bibliothèque appelle la première avec deux arguments et elle lève `less number of parameters` ; les gardes passent la position de départ, et le bloc aussi :

HARO

```
o1 = new stzList([ 1, 2, "♥", 4, 5, "♥", "♥" ])
? o1.FindNthNext(2, "♥", :StartingAt = 1)
#--> 6
? o1.FindNextNth(2, "♥", :StartingAt = 1)
#--> 6
? Q("---haro").NumberOfOccurrenceOfCharLeftSide("-")
```

```
#--> 3
? Q("---haro").HowManyOccurrenceOfCharStartSide("-")
#--> 3
o2 = new stzList([ "C", "B", "A" ])
o2.SwapItems(:AtPositions = 1, :And = 3)
? @@( o2.Content() )
#--> [ "A", "B", "C" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le diaporama promettait davantage. Une famille de cinq noms en `CharsRemoved`, pour retirer des caractères aux bouts, est nommée dans le diaporama. Deux sont définis, `LeadingCharsRemoved` et `TrailingCharsRemoved`. Trois ne le sont pas : `LeadingAndTrailingCharsRemoved`, `TrailingAndLeadingCharsRemoved` et `RepeatedCharsRemoved`. `RemoveLeadingAndTrailingChars` est défini, et n'est pas l'un des cinq.

Règle 9 : la précision plutôt que la généralité. Le nom précis dit ce qui est compté. Les caractères d'une chaîne sont comptés par `NumberOfChars`, et `Size()` est la longueur en octets de l'hôte. Les deux verbes de liste du même bloc montrent la règle sur une liste : le nom précis, `ContainsBoth` ou `ContainsAll`, dit ce qui est demandé. Les deux réponses diffèrent-elles sur un mot arabe ? Exécutez-le :

HARO

```
? Q("سلام").NumberOfChars()
#--> 4
? Q("سلام").NumberOfBytes()
#--> 8
? Q("سلام").Size()
#--> 8
o1 = new stzList([ "tea", "rice", "fish", "salt" ])
? o1.ContainsBoth("tea", "fish")
#--> TRUE
? o1.ContainsAll([ "tea", "oil" ])
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

L'exécution répond 4 à l'un et 8 à l'autre. Le mot a quatre lettres ! La propre ligne de documentation de la bibliothèque pour `NumberOfChars` liste *size* parmi ses mots alternatifs. L'exécution montre que `Size()` ne veut pas dire cela. Je le liste parmi les demandes.

Règle 7 : un préfixe `viz` montre la réponse sous le mot, et le suffixe `XT` ajoute les positions comme des nombres ; le chapitre 8 exécute le bloc, parmi les formes d'un nom.

La narration et le diaporama écrivent `vizFindAll` sur une chaîne. Il n'y existe pas à ce commit : mon exécution a répondu `Calling Method without definition: vizfindall`. `vizFind` existe. Une liste a son propre `VizFindAll`.

LES BUTS

Sous la grammaire il y a sept buts, écrits dans le diaporama de 2022 comme une matrice de trente-trois fonctionnalités, que la section 4 compte. Mon étude les lit comme le squelette architectural de la bibliothèque : une fonctionnalité qui n'en sert aucun est refusée, et un but sans fonctionnalité est un écart. Le résumé de l'étude est la lecture d'une session et non un texte de la bibliothèque, et aucun document de la bibliothèque n'énonce la règle en ces mots. Le plus proche est le document de conception du moteur : le moteur doit permettre les sept.

Je ne listerai pas les sept comme une table de slogans. Voici ce qui s'exécute de trois d'entre eux.

Flexibilité : l'ordre des paramètres nommés est libre. Deux appels, avec les paramètres dans deux ordres, sont-ils le même appel ?

HARO

```
o1 = new stzString("----@@--@@-------@@----@@---")
o1.ReplaceNextNthOccurrence(2, :Of = "@@", :StartingAt = 12, :With = "##")
? o1.Content()
#--> ----@@--@@-------@@----##---
o2 = new stzString("----@@--@@-------@@----@@---")
o2.ReplaceNextNthOccurrence(2, :With = "##", :StartingAt = 12, :Of = "@@")
? o2.Content()
#--> ----@@--@@-------@@----##---
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux ordres impriment `----@@--@@-------@@----##---` ! C'est l'ordre libre des paramètres nommés ; l'autre ordre libre du diaporama, un nombre et une chaîne permutés dans leurs positions, est celui du chapitre 8, et il ne s'exécute pas. Une tension doit être dite avec. La conception de Haro refuse les paramètres à ordre libre comme grammaire, et la bibliothèque les garde. Deux faits sont vrais aujourd'hui. Le bloc ci-dessus les exécute, donc ils sont dans la bibliothèque. La grammaire à laquelle une machine est tenue les laisse de côté, par le principe de Haro. Que la bibliothèque les garde au-delà de cette grammaire ou les retire, c'est une décision que le dossier ne tient pas. Je ne la prends pas ici. Ils sont gardés, et ils ne sont pas de la grammaire.

Cohérence : un vocabulaire à travers une chaîne et une liste. Le même verbe répond-il sur les deux ?

HARO

```
? @@( StzStringQ("hello").FindAllCS("L", :CaseSensitive = 0) )
#--> [ 3, 4 ]
? @@( StzListOfStringsQ([ "h", "e", "l", "l", "o" ]).FindAllCS("L", :CaseSensitive = 0) )
#--> [ 3, 4 ]
? Q("").IsEmpty()
#--> TRUE
? Q([]).IsEmpty()
#--> TRUE
```

✓ exécuté : a affiché ce que ce bloc promet

La perte à côté, du même groupe de diapositives : le conseil du diaporama d'employer le mot-clé Item avec tout type énumérable ne tient pas. `NumberOfItems()` sur une chaîne lève une erreur dans mon exécution : `Calling Method without definition: numberofitems`. Sur une liste il répond.

Maniabilité : un échec qui dit où, quoi, pourquoi, et que faire. La bibliothèque lève une erreur en quatre champs. Je n'imprime pas son message, parce que, dans mon exécution, sa première ligne portait un numéro de ligne de la bibliothèque, et un numéro de ligne n'est pas une promesse.

Les quatre autres buts (l'expressivité, la fiabilité, les métaphores humaines et les abstractions pratiques) je les laisse aux tables de l'étude. Je ne les ai pas exécutés pour ce chapitre, et je ne décrirai pas d'après le diaporama ce que je n'ai pas exécuté.

Vient ensuite la décision, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé un style qu'on aurait changé quand l'IA est devenue l'outil d'écriture principal. Le chapitre 5 raconte ce que j'ai trouvé, et ce que j'ai laissé aller. Le code d'implémentation que l'IA a écrit est accepté comme bon. Ce qui est exigé, c'est que chaque ligne qu'un programmeur, humain ou agent d'IA, est prié d'écrire avec Softanza embrasse le style historique.

J'ai refusé un but qui ne vit que dans un document. Les sept buts sont écrits dans un document de conception et dans des balises qu'aucun tribunal ne lit. Je n'ai trouvé de tribunal sur aucun des deux.

J'ai refusé de cacher le coût du style. L'étude a mesuré des marqueurs par millier de lignes. Les gardes narrées, les fichiers de test écrits pour être lus comme des articles, portent la promesse `#-->` à moins d'une ligne par millier. Elles portent un appel à la fonction d'aide d'assertion à quatre-vingt-cinq. Les tests numérotés plus anciens portent la promesse à cinquante-deux. Le code de la bibliothèque porte la promesse à moins de deux. Le style est le plus dense là où j'écrivais pour un lecteur, et plus mince là où l'IA écrivait pour d'autres ; l'y porter est l'étape suivante. Un marqueur est une lecture du texte, non un jugement du code ; l'étude le dit elle-même.

J'ai refusé de recompter les balises comme les miennes. Les formes des fonctions sont marquées dans le code par des info-balises, dans des commentaires qui commencent par `#<` et un `@`, appliquées à la main. L'étude y lit le vocabulaire, au niveau des fonctionnalités, de deux buts, l'expressivité et la flexibilité, et en a compté 6 090. J'ai recompté le 8 octobre 2026 et j'en ai trouvé 6 126, avec 89 orthographes. Mais 1 360 d'entre elles se trouvent hors des dossiers d'archive, et 4 766 dans les dossiers d'archive. Donc la phrase « appliquées à la main en 6 090 points » est, à ce commit, surtout une phrase sur l'archive. Aucune garde ne les lit : la couche de réflexion lit le nom d'une méthode, avec une grammaire, et non sa balise.

J'ai refusé le décompte comme un compte de réussites. L'étude compte trente-trois fonctionnalités : vingt-quatre construites, quatre transformées avec le projet, deux qui ont changé de forme, une partielle, une archivée, une seulement un nom. L'arithmétique tient. Mais c'est un compte de noms trouvés, et une définition trouvée n'est pas un test qui passe. Plusieurs des fonctionnalités sont nommées dans le diaporama et ne sont pas définies dans le code de la bibliothèque, l'archive mise à part. Ce sont les familles `FF` et `FX` pour les sections, les familles `dft` et `inf` pour les défauts et les infinis, `WalkFF` et `AddWalkerFF`, et le `Check` sur une liste. Ce dernier est le vérificateur du diaporama, qui répond `1` ou `0` pour chaque élément. Deux sont définies et ne concordent pas encore avec certains exemples du diaporama. `Yield` devrait donner le résultat d'une expression pour chaque élément. `Perform` devrait appliquer une action à chaque élément. J'imprime le décompte, et j'imprime ses pertes avec lui. Le diaporama dit aussi « 7 causes racines de la complexité de la programmation », à côté des sept buts. Aucune diapositive ne liste ces sept causes, et aucun autre fichier du dossier ne les nomme. Le chapitre n'en nomme aucune et ne les met pas en correspondance avec les buts, une par une.

Vient ensuite le tribunal, et je l'exécute.

5. LE TRIBUNAL

`coderule_narrated` prouve qu'une règle de la maison est une règle sur le graphe du code ; que le code planté est trouvé par son nom ; que le code propre passe ; et que la bibliothèque en liste quatorze. `testoor_promises_narrated` prouve que la promesse `#-->` est jugée comme l'affirmation d'un arrêt : tenue, divergente, non atteinte ou non jugée. `602_l_function` et `59_use_of_the_l_small_function` prouvent le langage d'intervalles. `527_identifying_lists_inside_a_string` prouve les listes hébergées dans des chaînes. `48_replacebymany` prouve que les remplacements se font dans l'ordre. `483_w_and_wf_forms_in_conditional_code` prouve la condition dans une chaîne, en deux écritures du même balayage. `28_numberofoccurrenceofcharleftside` prouve les noms alternatifs en arabe

comme en latin, et porte une note disant que la promesse de l'archive était fausse et que l'implémentation fait autorité. `646_content` prouve les paramètres nommés. `972_narration_viz-finding_a_recurring_substring` prouve la visite viz.

Voici le tribunal de la maison lui-même, les quatorze règles de la bibliothèque et non les quatorze du style. La coïncidence numérique est une coïncidence. Que répond-il à une faute d'orthographe plantée, et à une écriture plantée sur une constante ? Exécutez-le :

HARO

```
? @@( StzCodeRuleNames() )
#--> [ "no-len-method", "q-returns-object", "no-aggressive-verbs", "engine-first", "q-has-plain-
twin", "no-case-collision", "dead-knob", "setter-resets-on-reject", "setter-only-moves-one-way",
"misspelled-name", "library-prints", "empty-method-body", "writes-a-mutable-constant", "value-
called-as-function" ]
cOne = "class Inbox" + nl + "def Recieve(x)" + nl + "   return x" + nl
? StzCheckCode(cOne)[1][:rule]
#--> misspelled_name
? StzCodeIsClean(cOne)
#--> TRUE
cTwo = "class Inbox" + nl + "def Count()" + nl + "      nl = 3" + nl + "        return nl" + nl
? StzCheckCode(cTwo)[1][:rule]
#--> writes_a_mutable_constant
? StzCodeIsClean(cTwo)
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

La faute d'orthographe est trouvée par son nom, `misspelled_name`, et le code reste propre, `TRUE`, parce que c'est un avertissement. L'écriture sur une constante est trouvée par son nom, `writes_a_mutable_constant`, et le code est refusé, `FALSE`, parce que c'est une erreur. Le tribunal a refusé l'écriture sur une constante !

Un détail d'orthographe, pour l'annexe B : la liste des noms emploie des traits d'union, et le constat de huit des quatorze revient avec des soulignés. Le bloc le montre : la liste imprime `misspelled-name`, et le constat imprime `misspelled_name`.

Ce que le tribunal ne lit pas, c'est le code d'un programme écrit avec la bibliothèque. Les quatorze règles de la maison sont des règles du propre code de la bibliothèque ; une règle d'écriture pour le code écrit avec elle est conçue (section 7).

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`coderule_narrated` 77 assertions
base/test/meta/coderule_narrated.ring

`602_l_function` 6 assertions base/test/list/602_l_function.ring

`59_use_of_the_l_small_function` 8 assertions
base/test/list/59_use_of_the_l_small_function.ring

`527_identifying_lists_inside_a_string` 32 assertions
base/test/string/527_identifying_lists_inside_a_string.ring

`48_replacebymany` 1 assertions base/test/string/48_replacebymany.ring

`483_w_and_wf_forms_in_conditional_code` 2 assertions
base/test/list/483_w_and_wf_forms_in_conditional_code.ring

`28_numberofoccurrenceofcharleftside` 8 assertions
base/test/string/28_numberofoccurrenceofcharleftside.ring

`646_content` 1 assertions base/test/string/646_content.ring

`972_narration_viz-finding_a_recurring_substring` 7 assertions
base/test/string/972_narration_viz-finding_a_recurring_substring.ring

`testoor_promises_narrated` 46 assertions
base/test/testoor/testoor_promises_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en application par défaut. Un formateur s'exécute à chaque commit. Pour le code qu'un programmeur écrit avec Softanza, le tribunal qui appliquerait le style est conçu et pas encore construit, et le chapitre 5 le dit.

Le terrain mène en ayant ses buts quelque part où un lecteur regarde d'abord. Les sept buts sont dans un document de conception et dans des balises qu'aucun tribunal ne lit, et les notes de fonctionnement du dépôt propre de la bibliothèque n'en nomment aucun. Le récit d'un lecteur IA de la bibliothèque dit que soixante documents de paradigme pourraient aisément être soixante idées déconnectées. L'étude y lit un signe que la discipline est dans le code et écrite nulle part où un lecteur regarderait d'abord. C'est un document de session, et non un témoin indépendant.

Le site a montré un jour un dialecte réduit : ses blocs n'avaient ni paramètre nommé ni scène. Il a depuis mis les quatorze règles sur sa page Craft et les règles comme données. Je n'ai pas remesuré les blocs du site après cela.

Là où je pense que le style se distingue, c'est que chaque règle sert une étape de pensée, et que chaque étape a une forme écrite. La forme du verbe est un contrat sur l'objet. Et la promesse sous la ligne est l'exemple, le test et la documentation à la fois. La matrice de 2022 a servi de squelette à un architecte pour environ trois cent mille lignes écrites à la main, selon le compte de mon étude.

La limite, à la section suivante, dit ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT les quatorze règles de la maison de la bibliothèque comme un tribunal sur son propre code ; la promesse jugée comme une affirmation ; les petits langages des intervalles, des listes et des conditions dans des chaînes ; l'emplacement du code ; le préfixe viz ; les alternatives et les paramètres nommés dans n'importe quel ordre, en partie

CONÇU un tribunal qui juge le style d'un programme écrit avec la bibliothèque ; un tribunal qui lit les buts ; la grammaire d'écriture déclarée en Haro ; la

forme classique d'un bloc de test comme forme unique

ABSENT — toute machine qui juge le style du code que vous écrivez aujourd'hui ; l'écriture dans n'importe quelle langue humaine du nom d'une fonction ; la phrase quasi naturelle telle que le diaporama l'écrivait ; le code fondé sur eval sur le nom d'un paramètre

Ce qui est construit : les quatorze règles de la maison de la bibliothèque comme un tribunal sur son propre code, exécutées dans le bloc ci-dessus. Aussi construits : la promesse jugée par une machine ; les petits langages, l'emplacement, le préfixe. Et les fonctionnalités qui s'exécutent (`CS`, `XT` et `viz`, les paramètres nommés, le même verbe sur une chaîne et une liste ; `ZZ` s'exécute au chapitre 8).

Ce qui est conçu :

- **Un tribunal qui juge le style d'un programme écrit avec la bibliothèque.** C'est une spécification dans l'étude : les quatorze règles comme règles de la maison sur le graphe d'un extrait. Ses exemples sont un argument positionnel là où un argument nommé existe, ou une ligne imprimée sans promesse. Je l'ai cherché dans la bibliothèque : il n'est pas encore construit, c'est l'étape suivante.
- **Un tribunal qui lit les buts.** Il lirait les balises. À venir.
- **La forme classique du bloc de test comme forme unique des nouveaux blocs.** Le jugement de l'étude est que le retour « n'est pas une réécriture » mais une décision. La forme classique est la forme unique des nouveaux blocs, et les blocs convertis restent tels qu'ils sont. Le README de l'arbre de tests dit encore de la forme classique qu'elle est la surface de test faisant autorité. Je n'ai trouvé aucune décision qui en fasse la forme unique : cela reste une décision, et le chapitre 5 la nomme.
- **La grammaire d'écriture déclarée en Haro.** La moitié vocabulaire existe : les formes des noms de la bibliothèque sont des données en Haro, et le relevé de la session est de 584 orthographes livrées sur 595 qui concordent. La moitié usage, la scène, la chaîne, l'emplacement du code et la promesse, n'est pas déclarée.

- **La discipline déclarée une fois.** En partie faite sur le site, comme données, ce qui est la source de la page et non un tribunal.

Ce qui est absent :

- Toute machine qui juge, aujourd'hui, le style d'un programme écrit avec la bibliothèque ; à côté, le tribunal de la maison juge le propre code de la bibliothèque.
- Les règles 12 et 13 comme formes qui s'exécutent. Dans mes exécutions, les exemples quasi naturels du diaporama ne s'exécutaient pas tels qu'écrits, ni une fonction nommée en chinois ou en français.
- Le `@` sur le nom d'un paramètre comme forme qui fonctionne.
- La forme à accolades du langage d'intervalles, `L('{ "♥1" : "♥6" }')`, qui lève une erreur, et les intervalles qui changent de largeur, comme `day8 : day11`, qui lèvent aussi une erreur, avec une erreur de variable non initialisée de l'hôte. Le constructeur de liste à partir d'une chaîne en lève une aussi. Ces trois constats viennent de mes exécutions ; aucun bloc de ce chapitre ne les garde.
- La pile d'appels visualisée, et les décorateurs de cache et de journal, qui existent comme noms dans des commentaires.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Quatorze règles contre lesquelles vous pouvez tenir votre propre ligne. Élevez la valeur, demandez dans une scène, mettez la réponse sous `#-->`, nommez le paramètre, chaînez une étape par ligne, lisez la forme du verbe ; et mettez un intervalle, une condition ou une palette dans une chaîne.
L'analyste	Les mots de votre problème deviennent les noms des étapes. Une règle écrite dans une chaîne est une règle que vous pouvez stocker et lire, et non du code auquel vous devez faire confiance.
Le concepteur	Le préfixe viz dessine la réponse là où l'œil l'attend. Les buts sont les rubriques sous lesquelles une conception est jugée, une fonctionnalité à la fois.
Le décideur	Sept buts et trente-trois fonctionnalités sont une grille de lecture. Selon la lecture de l'étude, une fonctionnalité qui n'en sert aucun est refusée, et un but sans fonctionnalité est un écart. Le décompte est imprimé avec ses pertes.
Le formateur	La discipline s'enseigne entière et dans l'ordre. La garde convertie pour les listes dans des chaînes garde en commentaire l'espacement erroné de l'ancienne promesse, ce qui est une leçon sur le coût des réponses tapées à la main.
L'agent	Les règles vous sont données comme données sur le site. Pour le code que vous écrivez, le tribunal est conçu et pas encore construit. Écrivez la question, puis la promesse dessous, et mettez votre condition dans l'emplacement, non dans l'évaluation d'une chaîne.

9. EXERCICE

Écrivez, avant d'exécuter quoi que ce soit, la ligne dont chaque tâche a besoin et la promesse dessous :

1. À partir d'une chaîne, la liste `[ "♥7", "♥8", "♥9" ]`.
2. La chaîne `"haro php haro"` avec les deux `haro` numérotés `♥1` et `♥2`, en un seul appel qui emploie le petit langage de la tâche 1.
3. La liste `[ "C", "B", "A" ]` ordonnée en échangeant les premier et troisième éléments, avec les positions nommées.
4. Le nombre de caractères de "سلام", avec le nom qui dit ce qui est compté.
5. Une supposition : qu'imprime `L("day8 : day11")` ?

HARO

```
# 1.
# 2.
# 3.
# 4.
# 5.
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Pour la cinquième, vous aviez le droit de vous tromper, et vous vous êtes probablement trompé : l'intervalle change de largeur, alors `L` confie son texte à l'hôte pour l'évaluer, l'hôte lit `day8` comme une variable, et il lève une erreur, `Using uninitialized variable: day8`. C'est la limite du petit langage, apprise en étant surpris une fois. Ne changez pas votre promesse pour qu'elle corresponde à la sortie. Expliquez d'abord la sortie.

Je vous laisse celui-ci. Le premier appel du bloc de la règle 8, son départ nommé retiré, a deux arguments. Rien d'autre ne change :

HARO

```
o1 = new stzList([ 1, 2, "♥", 4, 5, "♥", "♥" ])
? o1.FindNthNext(2, "♥")
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez celui-ci et lisez ce qui revient. Puis réparez l'appel. La section 3 dit d'où vient cette forme d'appel et ce que font les gardes à la place. Quand il sera réparé, vous saurez pourquoi un appel que la narration même de la bibliothèque écrit lève une erreur, et ce que les gardes passent qu'il ne passe pas.

Le chapitre 8 prend le nom d'une fonction en entier, comme une expression linguistique, puis la portée : ce nom est une phrase, et ses préfixes, ses suffixes et ses paramètres sont une grammaire que la bibliothèque peut lire.

OÙ SONT LES PREUVES

NARRATION	`stz-functions-as-linguistic-expressions.md`
NARRATION	`stz-conditional-code-narration.md`
NARRATION	`stzlist-in-strings-narration.md`
NARRATION	`stzstring-vizfind-narration.md`
NARRATION	`stz-function-forms-narration.md`
SUR LE SITE	craft
SUR LE SITE	principles
DÉCISIONS DE L'ÉTUDE	12.2, 12.14, 12.16, 12.17, 12.21

PARTIE II · CHAPITRE 8

Les fonctions comme expressions linguistiques, et la portée

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 13 bloc(s) sur 13 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je lis le nom d'une fonction comme la bibliothèque le lit : une phrase dont les préfixes, les suffixes et les paramètres sont une grammaire. Je parcours les formes d'un nom qui tournent, et je compte les cinq qui ne tournent pas. Puis je passe à la portée : le même verbe, trois verdicts, avec le cadre dans le verbe et non dans un drapeau posé plus tôt. Je dis ce que la conception de Haro garde des formes, ce qu'elle refuse, et ce que le tribunal prouve. Le principe est celui-ci : le nom d'une fonction est une phrase, et ses préfixes, ses suffixes et ses paramètres sont une grammaire que la bibliothèque peut lire.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Sur le terrain, le nom d'une fonction est une étiquette. Il suit une convention, camel case ou snake case, et il vous dit à peu près ce que fait la fonction ; les détails sont dans la signature et la documentation. Un nom ne porte pas de grammaire. Les outils voisins ne parlent que de structure. Python et Swift offrent l'argument nommé, pour qu'un appel puisse dire quel paramètre est lequel. L'interface fluide enchaîne les appels pour qu'une ligne se lise comme une phrase, et la surcharge donne à un seul nom plusieurs signatures.

Et quand le terrain veut qu'une fonction se comporte autrement dans un autre contexte, il change son sens par un drapeau posé ailleurs. Le drapeau est un mode, une locale, une variable d'environnement. Le nom de la fonction ne vous dit pas

dans quel cadre elle va s'exécuter.

2. LE PREMIER PRINCIPE

Le nom d'une fonction est une phrase, et ses préfixes, ses suffixes et ses paramètres sont une grammaire que la bibliothèque peut lire. L'article qui a le premier avancé cette thèse dit, dans les mots que cite l'étude de ses racines, qu'elle a été obtenue « without machine learning or NLP systems ». Elle l'a été par des conventions de nommage qui encodent le sens, un ordre des paramètres qui épouse la parole, des formes souples et des valeurs par défaut intelligentes.

3. LA FORME SOFTANZA

Regardez un nom comme la bibliothèque le regarde. `FindNextSTCS` : trouver le suivant, à partir d'une position de départ, avec un drapeau de casse. `RemoveManyCS` : retirer, plusieurs choses, un drapeau de casse. Chaque suffixe est petit et ne veut dire qu'une chose, et, dans ces noms, l'ordre des suffixes est l'ordre des paramètres. Ici le premier nom trouve. Que laisse le second quand il retire toutes les occurrences de trois lettres, quelle que soit leur casse ? Exécutez-le :

HARO

```
o1 = new stzString("123456789HARO")
? o1.FindNextSTCS("haro", 5, FALSE)
#--> 10
o2 = new stzString("SOOooooFFfffTANNnnnZA")
o2.RemoveManyCS([ "o", "f", "n" ], FALSE)
? o2.Content()
#--> STAZA
```

✓ exécuté : a affiché ce que ce bloc promet

L'article promettait `SOFTANZA` sur la deuxième ligne. L'exécution imprime `STAZA` ! Le code a raison et l'article a tort : un retrait qui ne se soucie pas de la casse retire aussi les majuscules. J'imprime l'exécution.

La bibliothèque se contente-t-elle de suivre cette grammaire, ou la lit-elle ? Son analyseur de noms, `_StzParseName`, est un assistant interne que nulle garde ne couvre ; je l'appelle en premier ci-dessous, pour la lecture brute. La voie publique est

`ExplainMethod`, qui explique une méthode dans ces termes :

HARO

```
? @@( _StzParseName("FindNextSTCS") )
#--> [ "find next", [ "case-sensitive", "from-position", "active" ] ]
? @@( _StzParseName("RemoveManyCS") )
#--> [ "remove", [ "case-sensitive", "plural", "active" ] ]
? StzSelfDocQ("stzString").ExplainMethod("RemoveMany")
#--> RemoveMany -- Remove every occurrence of each given substring (mutating).
#-->   (plural form -- applies the operation to many items at once)
#-->   reads as: remove -- over many values at once
```

✓ exécuté : a affiché ce que ce bloc promet

Elle la lit. Voilà la thèse rendue mécanique : la grammaire est dans le code et pas seulement dans l'article. La ligne imprimée `reads as: remove -- over many values at once` est la lecture que l'analyseur fait de `RemoveMany`. L'analyseur décode les préfixes et les suffixes, les mots « except » et « is not », et la voix : un nom qui finit en `ed` est passif.

Je vais maintenant parcourir les formes. Le chapitre 5 en a montré quatre : l'active, la passive, la fluide et le paramètre nommé. Voici les autres, celles qui tournent.

Z et ZZ disent si une position arrive comme un nombre ou comme une section. Un suffixe, et la réponse change de forme :

HARO

```
o1 = new stzString("haro php haro ruby haro")
? @@( o1.FindZ("haro") )
#--> [ 1, 10, 20 ]
? @@( o1.FindZZ("haro") )
#--> [ [ 1, 4 ], [ 10, 13 ], [ 20, 23 ] ]
o2 = new stzString("SOFTANZA")
? @@( o2.FirstHalfZ() )
#--> [ "SOFT", 1 ]
? @@( o2.FirstHalfZZ() )
#--> [ "SOFT", [ 1, 4 ] ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le préfixe viz change la sorte de réponse : un dessin. La position est montrée sous le texte, et une option peut la numéroter :

HARO

```
o1 = new stzString("fjharoljharodjharog")
? o1.vizFind("haro")
#--> fjharoljharodjharog
#--> --^-----^-----^----
? o1.vizFindXT("haro", [ :Numbered = TRUE ])
#--> fjharoljharodjharog
#--> --^-----^-----^----
#-->   3     9     15
```

✓ exécuté : a affiché ce que ce bloc promet

Le pluriel étend un verbe à plusieurs choses sans boucle, et **la forme d'exception** ne garde que ce que vous nommez. L'exemple de l'article pour la seconde, `RemoveNonLettersExcept`, n'existe pas. La forme existe, sous d'autres noms :

HARO

```
Q("__H~~~AR***O__") {
        ? @@( FindMany([ "~", "*" ]) )
        #--> [ 4, 5, 6, 9, 10, 11 ]
        RemoveMany([ "_", "~", "*" ])
        ? Content()
        #--> HARO
}
o2 = new stzString("--Haro--&__Softanza__")
o2.RemoveAllExcept([ "Haro", "&", "Softanza" ])
? o2.Content()
#--> Haro&Softanza
```

✓ exécuté : a affiché ce que ce bloc promet

Le négatif est un mot dans le nom, pas un NOT autour de l'appel :

HARO

```
? Q("*").IsNotLetter()
#--> TRUE
o3 = new stzList([ "H", "A", "R", "O" ])
```

```
? o3.IsNotAString()
#--> TRUE
? o3.DoesNotContain("♥")
#--> TRUE
```

✓ exécuté : a affiché ce que ce bloc promet

La forme profonde répond par un chemin, une liste d'indices qui va aussi profond que les données, et la même famille remplace et retire à cette profondeur, avec `:By` pour emplacement nommé là où le `Replaced` du chapitre 5 disait `:With` ; les verbes profonds acceptent les deux :

HARO

```
Q([ "you", "other", [ "other", "you", [ "you" ], "other" ], "other", "you" ]) {
        ? @@( DeepFind("you") )
        #--> [ [ 1 ], [ 3, 2 ], [ 3, 3, 1 ], [ 5 ] ]
        DeepReplace("you", :By = "♥")
        ? @@( Content() )
        #--> [ "♥", "other", [ "other", "♥", [ "♥" ], "other" ], "other", "♥" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

La forme immuable, QC, exécute toute la chaîne sur une copie. Après cette chaîne, que contient l'original ?

HARO

```
o1 = new stzString("haxxrxo")
? o1.RemoveQC("x").UppercaseQ().Spacified()
#--> H A R O
? o1.Content()
#--> haxxrxo
```

✓ exécuté : a affiché ce que ce bloc promet

La chaîne a imprimé `H A R O`, et l'original contient encore `haxxrxo` !

La forme mal orthographiée est acceptée, avec une note sur ce qui la soutient. Une faute d'orthographe dans un nom fonctionne :

HARO

```
? Q("   Haro ").WithoutSapces()
#--> Haro
```

✓ exécuté : a affiché ce que ce bloc promet

L'article dit que la bibliothèque reconnaît et corrige les fautes de frappe mineures, et que l'intelligence réside dans la bibliothèque elle-même. Le code montre comment la forme fonctionne : `WithoutSapces` fonctionne parce que cette orthographe a été écrite à la main comme un alias. Une autre faute, `WithoutSpces`, échoue. La forme est donc réelle, et dans ce que j'ai suivi c'est une liste d'alias déclarés ; le moteur de correction que décrit l'article ne fait pas partie de ce que j'ai suivi. Je n'ai pas suivi tous les fichiers qui parlent de correction ; un message de la classe objet ne fait que suggérer un nom, avec « Did you mean ».

Voici maintenant la moitié du chapitre qui concerne la portée. Le même défaut apparaît partout en programmation : un comportement gouverné par un cadre invisible. Je tiens cela pour une vue, non pour un résultat ; ce qui la soutient ici, ce sont trois instances : les verbes de l'expression régulière, le système et le placement d'un déploiement. Une fonction fait une chose ici et une autre là, et rien dans son nom ne dit dans quel cadre elle se trouve. La réponse, en cinq gestes, est le paradigme que j'appelle la programmation orientée portée. Quatre viennent d'abord. Nommer le cadre caché. En faire un petit ensemble fermé de portées nommées. Choisir la portée à l'endroit de l'appel, dans le verbe, et pas dans un drapeau posé plus tôt. Et donner à la bibliothèque la portée pour raisonner, une prise sémantique et pas un commentaire.

Il est né dans les expressions régulières, où le cadre est la part du texte qu'un motif doit couvrir. Le même motif, le même texte, trois verbes : que répond chacun ?

HARO

```
cText = "First: Line1" + nl + "Second: Line2" + nl + "Third: Line3"
o1 = new stzRegex("^Second:.*$")
o1 {
        ? Match(cText)
        #--> FALSE
```

```
        ? MatchFirst(cText)
        #--> FALSE
        ? MatchLine(cText)
        #--> TRUE
}
```

✓ exécuté : a affiché ce que ce bloc promet

`Match` répond `FALSE` : il demande si le motif couvre la valeur entière. `MatchFirst` répond `FALSE` aussi : il demande si le motif apparaît quelque part, avec les ancres de ligne désactivées. `MatchLine` répond `TRUE` : il fait que `^` et `$` s'accrochent à chaque ligne. Le cadre est dans le verbe. Il y a aussi une portée de mot, qui fournit les limites de mot pour que vous ne les écriviez pas. Elle réécrit pour de bon le motif de son objet, c'est pourquoi la deuxième instance est un nouvel objet avec le même motif. Le même motif répond-il pareil sous `MatchFirst` et sous `MatchWord` ?

HARO

```
o2 = new stzRegex("pre\w+")
? o2.MatchFirst("compress")
#--> TRUE
o3 = new stzRegex("pre\w+")
? o3.MatchWord("compress")
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

Le même motif répond `TRUE` sous `MatchFirst` et `FALSE` sous `MatchWord` !

La deuxième instance est le système. Le code d'une fonctionnalité s'écrit dans une portée nommée, et le même verbe reçoit un verdict différent dans chacune. La portée peut faire se comporter un portable comme un appareil qu'il n'a pas, ou refuser ce que la cible interdit. Ici, une solution avec un backend sur un serveur Linux et un firmware sur une petite carte. Que répond chaque système quand on lui demande le même verbe, `Spawn` ? Exécutez-le :

HARO

```
oSol = new stzPlatformProfile("my-iot-product")
oSol {
        DevelopedOn(:Windows)
```

```
        WithServer(:backend, :LinuxServer)
        WithApp(:firmware, :ESP32)
        oBack = App(:backend).System()
        oFirm = App(:firmware).System()
}
? oBack.Spawn("worker")
#--> native
? oFirm.ReadPin(4)
#--> rehearsed
try
        oFirm.Spawn("worker")
catch
        ? cCatchError
done
#--> scope 'esp32' (espidf) forbids 'process' -- spawn 'worker'. This target has no such
capability.
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le même verbe, `Spawn`, a deux verdicts ! Le serveur répond `native` ; la carte répond `scope 'esp32' (espidf) forbids 'process'`. C'est la portée qui décide, pas l'opération. Et le portable n'a pas de broche à lire, alors la portée répond `rehearsed`. C'est le cinquième geste : quand les capacités d'une portée diffèrent de celles de l'hôte, la bibliothèque comble l'écart, dans les deux sens. Deux systèmes déclarés peuvent être comparés comme des données, avec un ensemble fermé de capacités :

HARO

```
oEsp = DeclareSystem("esp32-firmware")
oEsp.SetOSName(:rtos)
oEsp.SetCapabilityList([ :gpio, :network, :clock, :threads ])
oBox = DeclareSystem("laptop")
oBox.SetOSName(:windows)
oBox.SetCapabilityList([ :filesystem, :process, :network, :clock, :threads ])
? @@( oBox.Forbids(oEsp) )
#--> [ "filesystem", "process" ]
? @@( oBox.Requires(oEsp) )
#--> [ "gpio" ]
```

✓ exécuté : a affiché ce que ce bloc promet

L'exécution a imprimé `[ "filesystem", "process" ]` pour ce que le portable peut faire et que la carte ne peut pas, et `[ "gpio" ]` pour ce dont la carte a besoin et que le portable n'a pas. C'est la deuxième instance. La troisième est le déploiement : la cible est le cadre, et la portée décide de ne pas livrer ce que la cible fait déjà bien ; le chapitre 4 en a montré le plan. La quatrième est la concurrence, et c'est un plan.

La conception vient ensuite, avec ce que Haro garde de ces formes et ce qu'il refuse.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Dans la conception de Haro, le principe 7 fait des formes une grammaire que le langage possède. Elle est conçue et non construite : les formes sont déclarées une fois, et le compilateur générerait et vérifierait chaque famille. La forme passive ne peut pas muter ; un verbe avec Q renvoie l'objet ; l'ordre des suffixes est l'ordre des paramètres ; un W prend un littéral analysé à la compilation. Le prototype ne juge que des noms : il ne lit pas les corps et ne juge pas l'ordre des suffixes. Sa première mesure est la classe chaîne de la bibliothèque : parmi ses 595 orthographes livrées des verbes find, replace et remove, 584 s'accordent avec la déclaration et 11 non. J'ai dit au chapitre 5 ce que Haro a refusé comme grammaire : la forme mal orthographiée, l'ordre libre des paramètres et les noms de méthodes par langue. Voici ce que cela veut dire pour ce chapitre : cinq formes ne tournent pas.

La forme future a été écourtée exprès. L'en-tête du fichier de code naturel la date du 10 juillet 2026 : la file globale d'actions différées est partie, et ses enveloppes FQ et FFQ avaient déjà disparu. La raison est écrite dans la conception : le langage naturel exige de bâtir tout le modèle de calcul avant d'exécuter, et le plan de jetons du moteur le fait désormais. C'est un plan du moteur naturel et plus un suffixe.

La forme conditionnelle a changé de mécanisme. La narration dit qu'elle ne passe plus par `eval()` mais par un petit langage textuel que le moteur compile, et que l'ancienne forme étendue, WXT, est maintenant un alias obsolète de W.

Pour la portée, le document de conception refuse le courant flottant. Le code d'une fonctionnalité n'est jamais écrit contre un « système courant » ambiant. Il nomme sa portée. Et pour la concurrence le document consigne deux refus avant qu'il y ait le moindre code. L'un est une méthode par opération qui étale le cadre sur de nombreux verbes, comme `DoubleEachConcurrently`. L'autre est un mode posé plus tôt. Ce mode est l'antimodèle contre lequel ce paradigme a été fondé. Il refuse

aussi une portée qu'aucun hôte ne peut exécuter. Sa première idée pour la concurrence était un vocabulaire de fils d'exécution, et le langage en dessous n'a pas de fils.

J'ai refusé de livrer tout le moteur en périphérie. La règle de placement est : ne livrer que ce qui est critique et soit propre à Softanza, soit faible sur la cible. Une cible qui excelle déjà dans une chose n'en reçoit rien.

Voici maintenant le tribunal, et les gardes qui prouvent chacune de ces affirmations.

5. LE TRIBUNAL

`00_regex_narrated` prouve les verbes `Match` et `MatchFirst` : `Match` ancre, `MatchFirst` cherche, Z rend les positions et ne les imprime pas. Elle n'affirme rien pour `MatchLine` et `MatchWord` : les blocs plus haut les montrent, et seuls des blocs classiques, qui ne sont pas des gardes, les couvrent. `972_narration_viz-finding_a_recurring_substring` prouve la réponse dessinée. `full_scope_model_narrated` prouve la portée du système, avec ses trois verdicts. `system_scope_model_narrated` prouve l'ensemble fermé de capacités ; l'une de ses assertions est une limite de temps, et la section 7 dit ce qu'une sonde en a montré. `delivery_narrated` prouve le plan de placement.

Où le terrain fait mieux vient ensuite.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`00_regex_narrated` 28 assertions
base/test/regex/00_regex_narrated.ring

`972_narration_viz-finding_a_recurring_substring` 7 assertions
base/test/string/972_narration_viz-finding_a_recurring_substring.ring

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

`system_scope_model_narrated` 62 assertions
base/test/system/system_scope_model_narrated.ring

`delivery_narrated` 34 assertions
base/test/system/delivery_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en convention. Un nom qui est une étiquette est plus facile à enseigner et plus facile à taper. La grammaire de ce chapitre doit s'apprendre, et certaines de ses formes sont rarement utilisées.

Le terrain mène en garantie. Les arguments nommés de Python sont vérifiés à l'appel, et les étiquettes de Swift par le compilateur, pour chaque fonction, dès le premier jour. Les formes d'ici sont jugées par le prototype de Haro, sur les noms et non sur les corps, et par l'analyseur propre de la bibliothèque, et seulement dans les noms qui les suivent.

Sur la portée, le terrain mène en maturité. La pratique la plus proche est le contexte explicite : l'habitude de Go de passer un contexte à chaque appel, un objet de locale qui porte un cadre. Elles sont plus anciennes et plus largement employées. Ce que ce paradigme ajoute, c'est que le cadre fait partie d'un petit ensemble fermé, choisi dans le verbe, et que la bibliothèque peut raisonner avec lui.

Enfin, la limite, avec ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT	quatorze des vingt-deux formes, certaines avec d'autres exemples que ceux de l'article, et trois autres sous une autre forme ; la bibliothèque qui lit un nom comme une phrase ; la portée dans les verbes de l'expression régulière, dans le système et dans le placement d'un déploiement
CONÇU	les formes déclarées une fois comme données et générées, en Haro ; le paradigme de portée pour le temps, les unités, la mémoire et le style
ABSENT	cinq formes : partielle, future, libre, par défaut et multilingue ; la concurrence comme portée : un plan sans code

Ce qui est construit : quatorze des vingt-deux formes d'un nom tournent, certaines avec d'autres exemples que ceux de l'article, et trois autres tournent sous une autre forme. La bibliothèque lit un nom comme une phrase. Et la portée tourne dans les verbes de l'expression régulière, dans le système et dans le placement d'un déploiement. L'article compte vingt-trois sections : les vingt-deux formes d'un nom, et « small functions », qui sont des fonctions et non une forme de nom. Le nombre du livre est vingt-deux, les formes d'un nom.

Ce qui est conçu : les formes de Haro comme données, avec des familles générées ; et le paradigme de portée pour le temps, les unités, la mémoire et le style, comme lignes d'un tableau.

Ce qui est absent, et la page doit dire chacune avec ce qui existe à côté :

- **Partielle**, `@Removed`, n'est que dans l'archive : l'appel n'est pas défini.
- **Future**, `BeforeQ`, a été écourtée exprès, comme je l'ai dit.

- **Libre**, le suffixe qui laisse un appel donner ses paramètres dans n'importe quel ordre et n'importe quel sous-ensemble : il n'y en a aucune définition hors de l'archive. L'analyseur propre de la bibliothèque connaît le nom `FX` et pas `FF`. Les deux narrations qui la racontent emploient deux noms, et le code n'a ni l'un ni l'autre.
- **Par défaut**, `dftRange` et `infRange` : non définis ; le commentaire de la bibliothèque appelle la forme informante pas encore construite.
- **Multilingue**, une méthode nommée en français ou en arabe : absente de la bibliothèque vivante. Les noms existent dans l'archive. Ce qui s'en approche le plus aujourd'hui, ce sont les paquets de langue du chapitre 11.
- Les trois qui tournent sous une autre forme. La forme d'exception existe, mais l'exemple de l'article n'existe pas. L'ordre libre, au sens de l'article, un nombre et une chaîne permutés dans leurs positions, ne fonctionne que pour le premier ordre : dans ma sonde du 8 octobre, l'appel permuté a terminé le programme avec un statut de un et n'a rien imprimé. L'ordre libre des paramètres nommés est autre chose, et le chapitre 7 l'exécute. Et la forme d'énoncé est à moitié en place : la moitié « All » fonctionne, et la moitié « No » est l'étape suivante. Le propre test classique de la bibliothèque promet vrai sur deux de ses quatre lignes « No », et l'exécution imprime faux sur les quatre.

Du côté de la portée : les verbes de l'expression régulière diffèrent du tableau du document de conception de la portée en deux endroits. `MatchLine` laisse un point traverser une ligne, là où le tableau dit qu'il reste dans une seule, et `MatchOneSegment`, que le tableau propose pour le cas paresseux, n'est pas défini. La portée du système est construite, et le document de conception l'appelle encore conçue : le dossier se contredit lui-même. Le document de conception de la portée, version 1.2 du 16 août 2026, est cité ici comme le texte du document. Le dossier ne dit pas qui l'a écrit, et je ne l'imprime pas comme mien. Le document de fondation du système, du 20 juillet 2026, dit que la fondation est implémentée. Lors d'une exécution de sonde sous charge, que les données du livre ne gardent pas, 61 des 62 assertions de sa garde narrée ont passé, et la seule assertion qui n'a pas tenu était une limite de temps. Je dis ce que le code fait : la portée du système est construite,

avec ce résultat imprimé à côté, et le mot « conçue » du document de conception est celui du document, périmé. Et la concurrence est un plan du dossier sans code : le mot `Concurrently` n'a aucune définition.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Lisez un nom comme une phrase : ses suffixes sont ses paramètres, dans l'ordre.
L'analyste	Une position peut arriver comme un nombre ou comme une section ; un nom dit laquelle.
Le concepteur	Le préfixe change la sorte de réponse : viz la dessine.
Le décideur	Une portée est choisie dans le verbe, donc le cadre est visible là où l'appel est fait, et une revue peut le voir.
Le formateur	La bibliothèque analyse un nom en sa phrase, de sorte qu'un apprenant peut lui demander ce que dit une méthode.
L'agent	Écrivez la portée dans le verbe. Un appel qui nomme une portée n'est pas déplacé dans un autre cadre par un drapeau posé ailleurs.

9. EXERCICE

Prenez un verbe d'une bibliothèque que vous utilisez et qui se comporte autrement dans deux cadres : un mode, une locale, une cible. Dites en une phrase quel est le cadre caché. Puis écrivez, comme vous voudriez qu'ils soient, deux noms pour le même acte dans les deux cadres, avec le cadre dans le verbe. Avant de rien exécuter dans Softanza, prenez `Match` et `MatchLine` sur un texte de trois lignes et écrivez ce que chacun imprime pour un motif qui s'ancre au début.

Exécutez-le. Si vos deux noms ne diffèrent que par un drapeau, vous n'avez pas encore déplacé le cadre dans le verbe.

Je vous laisse celui-ci, exprès. À la section 3, l'article promettait `SOFTANZA` à partir d'un retrait, et l'exécution a imprimé `STAZA`. Voici ce bloc avec un seul changement, le dernier argument de l'appel :

```
HARO
```

```
o2 = new stzString("SOOooooFFfffTANNnnnZA")
o2.RemoveManyCS([ "o", "f", "n" ], TRUE)
? o2.Content()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Mettez ce qu'il imprime à côté de ce que l'article promettait et à côté de ce que la première exécution a imprimé. La section 3 dit pourquoi la première exécution n'a pas donné la ligne de l'article ; je n'imprime pas ce que celui-ci donne. Quand vous l'aurez résolu, vous saurez ce que le drapeau de casse décide dans un retrait, et s'il suffit à lui seul à tenir la promesse de l'article.

Le chapitre 9 passe aux motifs de pensée. Il présente le Softanzuter, une représentation computationnelle d'une machine pensante. Il est fondé sur un ou plusieurs Regexuters, qui reconnaissent des motifs de pensée et y réagissent. Avec lui viennent Reaxis et les moteurs.

OÙ SONT LES PREUVES

NARRATION	`stz-functions-as-linguistic-expressions.md`
NARRATION	`stz-function-forms-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
SUR LE SITE	craft
SUR LE SITE	innovations
DÉCISIONS DE L'ÉTUDE	12.3, 12.13, 12.19, 12.21

PARTIE II · CHAPITRE 9

Les motifs de pensée : le Softanzuter, Reaxis et les moteurs

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 8 bloc(s) sur 8 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends tour à tour les trois sujets du titre : le Softanzuter, Reaxis et les moteurs, dont la ligne de temps et la grille tournent ici. Le Regexuter lie un motif à du code qui se déclenche et se souvient de son pourquoi ; Reaxis est de la programmation réactive au vocabulaire repensé. Je gravis l'échelle que dessine l'architecture, quatre échelons, et au troisième je lance le bloc qui échoue à enchaîner plutôt que de revendiquer un échelon qui n'est pas construit. Je dis ce que j'ai refusé : l'expression régulière comme simple façon de lire du texte, les mots du terrain pour la programmation réactive, les quatre frères, et la cascade. Tout sert une seule phrase : un Softanzuter est une représentation computationnelle d'une machine pensante, fondée sur un ou plusieurs Regexuters, qui lui permet de reconnaître des motifs de pensée et d'y réagir.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a trois outils pour faire réagir un programme. Les expressions régulières reconnaissent une forme dans un texte, et rien de plus : elles disent « cela correspond », et elles s'arrêtent. La programmation réactive, la famille des observables, des abonnés et des ordonnanceurs que Rx a rendue populaire, laisse les données circuler dans un programme au fil du temps. Son vocabulaire vient de la plomberie, de la téléphonie et de l'édition : tuyaux, contre-pression, sources chaudes et froides. Et les moteurs de règles, les systèmes de production du terrain avec leur

chaînage avant, déclenchent des règles quand des faits apparaissent, et continuent de déclencher tant que de nouveaux faits apparaissent ; c'est une connaissance générale, et les moteurs de règles du terrain sont plus anciens et plus mûrs que tout ce qui est dans ce chapitre.

Chaque outil est séparé. Une expression régulière ne déclenche pas de calcul, un flux réactif ne reconnaît pas de motif, et un moteur de règles n'a pas de vocabulaire qu'une personne emploierait pour dire une pensée. Je tiens cela pour une vue, non pour un résultat.

La section suivante donne la seule phrase par laquelle je réponds.

2. LE PREMIER PRINCIPE

J'ai donné à ce paradigme une seule phrase, et tout ce qui suit tente de la rendre mécanique : **un Softanzuter est une représentation computationnelle d'une machine pensante, fondée sur un ou plusieurs Regexuters, qui lui permet de reconnaître des motifs de pensée et d'y réagir.**

La section suivante montre jusqu'où la bibliothèque l'a rendue mécanique.

3. LA FORME SOFTANZA

L'architecture la dessine comme une échelle à quatre échelons, chacun ajoutant une chose.

Le premier échelon est le motif. Vous le verrez au chapitre 15 : une forme dans un milieu, une chaîne, une liste, un nombre, qui peut dire « cela correspond, et voici comment », et rien de plus. Le deuxième échelon donne une réaction à un motif : le Regexuter, un arc réflexe. Vous liez un motif à du code, vous traitez un texte, et le code se déclenche à chaque correspondance. Un Regexuter réagit, et il se souvient de ce qu'il a fait. Ici il masque les secrets d'une phrase. Que dit l'état qu'il garde sur chacun : ce qu'il a trouvé, ce qu'il a calculé, où, et quel déclencheur s'est déclenché ?

HARO

```
rxu() {
        Trigger(:SensitiveData = "(password|secret|key):\s*(\w+)")
        @C(:When = :SensitiveData, :Do = '{ @value = "REDACTED-" + len(@value) }')
        Process("User credentials: password: abc123 secret: def456")
```

```
        ? @@( Results() )
        #--> [ "REDACTED-16", "REDACTED-14" ]
        ? @@( Positions() )
        #--> [ 19, 36 ]
        ? State()[1][:matchedvalue]
        #--> password: abc123
        ? State()[1][:computedvalue]
        #--> REDACTED-16
        ? @@( State()[2][:dependson] )
        #--> [ "sensitivedata" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'état est une provenance : chaque réaction qui change une valeur garde son pourquoi. La dernière ligne, `[ "sensitivedata" ]`, est aussi une limite honnête, et j'y reviendrai : ce que le Regexuter enregistre comme dépendance n'est aujourd'hui que son propre déclencheur.

Voici maintenant le troisième échelon, celui dont parle la définition. Ce sont plusieurs Regexuters, sur des milieux différents, partageant un seul état, de sorte que ce qu'une faculté écrit est un milieu sur lequel une autre faculté peut chercher des correspondances. Une pensée est une entrée d'état avec sa provenance, et une pensée stabilisée est un état où plus rien ne se déclenche. Cet échelon n'est pas construit, et le bloc montre comment je le sais. Le second déclencheur ci-dessous est écrit pour les nombres de trois chiffres ou plus. Le premier déclencheur transforme le 7 en 700, qui en est un. Si la machine pensait en cascades, le second se déclencherait-il sur le résultat du premier ? Exécutez-le :

HARO

```
rxu() {
        Trigger(:Number = "\d+")
        Trigger(:Big = "\d{3,}")
        @C(:When = :Number, :Do = '{ @value = @number(@value) * 100 }')
        @C(:When = :Big, :Do = '{ @value = "BIG" }')
        Process("Found 7 items")
        ? @@( Matches() )
        #--> [ "7" ]
        ? @@( Results() )
        #--> [ 700 ]
        ? @@( Triggers() )
```

```
        #--> [ "number" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il ne l'a pas fait : les résultats disent `[ 700 ]`, les déclencheurs qui se sont déclenchés disent `[ "number" ]`. `BIG` n'apparaît nulle part ! Chaque déclencheur est confronté une seule fois au texte d'origine, et ce qu'un déclencheur écrit n'est pas offert aux autres. C'est toute la différence entre le deuxième échelon et le troisième.

La cascade tourne un échelon plus haut, au quatrième, l'agent d'IA : la machine pensante dotée d'un corps dans un monde, et d'un but. Son esprit est fait de compétences, chacune avec une condition et une action, qui s'enchaînent. La deuxième compétence se déclenche sur le fait que la première a écrit. `Pursue()` renvoie combien de déclenchements de compétences ont été vérifiés, et continue jusqu'à ce que le monde cesse de changer, ou vingt tours au plus. Que répond-il quand on le redemande ?

HARO

```
oAgent = new stzPIAgent("cook")
oAgent.MemoryQ().Learn("order", "state", "placed")
s1 = new stzAgentSkill("cook")
s1.SetWhen(func oM { return oM.Fact("order", "state", "placed") })
s1.SetDoes(func oM { oM.Forget("order","state","placed") oM.Learn("order","state","cooking")
return 1 })
s2 = new stzAgentSkill("serve")
s2.SetWhen(func oM { return oM.Fact("order", "state", "cooking") })
s2.SetDoes(func oM { oM.Forget("order","state","cooking") oM.Learn("order","state","served")
return 1 })
oAgent.AddSkill(s1)
oAgent.AddSkill(s2)
? oAgent.Pursue()
#--> 2
? oAgent.MemoryQ().Fact("order", "state", "served")
#--> TRUE
? oAgent.Pursue()
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond 2, puis 0, et entre les deux le fait `served` dit `TRUE`. C'est un point fixe. La garde qui le prouve étiquette la scène dans ses propres mots : `the cascade to fixpoint (Softanzuter at agent level)`, soit la cascade jusqu'au point fixe, le Softanzuter au niveau de l'agent. Au chapitre 17, vous verrez un agent de cette même classe refusé à une barrière. Voici son autre face : l'agent qui enchaîne, parce que son esprit est programmatique.

Le mot a donc trois sens dans le dossier, et le lecteur mérite qu'on lui dise lequel est lequel. D'abord, dans ce chapitre, « Softanzuter » est le troisième échelon, la machine pensante de la définition : conçu, pas construit. Ensuite, le moteur a un module du même nom, que le macroplan présente comme le socle des agents d'IA. Ses emplacements ont un état et une boîte aux lettres bornée, et un message envoyé dans une file pleine est refusé, compté et jamais jeté en silence ; l'agent du bloc ci-dessus, `stzPIAgent`, ne l'appelle jamais. Enfin, la conception d'un moteur universel derrière tout cela, un milieu où n'importe quel langage de motifs devient un calcul, est planifiée.

Reaxis est le deuxième sujet du titre du chapitre, et c'est une idée différente. C'est de la programmation réactive avec un vocabulaire repensé depuis la base. Une chose réactive est un Container, un Stream et une Rfunction : une fonction en attente, qui n'est pas un rappel. Les données circulent dans un flux en mots simples, une transformation, un filtre, ce qu'il faut faire de ce qui passe. Quatre prix entrent, 80, 90, 100 et 120 : lesquels ressortent, et à quelle valeur ?

HARO

```
aSeen = []
Rs = new stzReactiveSystem()
Rs {
        oStream = CreateStream("prices")
        oStream {
                Transform(func price { return price * 1.20 })
                Filter(func price { return price >= 100 })
                OnPassed(func item { aSeen + item })
                FeedMany([ 80, 90, 100, 120 ])
        }
        RunLoop()
}
? @@( aSeen )
#--> [ 108, 120, 144 ]
```

✓ exécuté : a affiché ce que ce bloc promet

`aSeen` dit `[ 108, 120, 144 ]` : le 80 est devenu 96 et a été écarté par le filtre ; le 90 est devenu 108 et est passé. Trois fonctions sont en attente, et rien de tout cela ne s'appelle un abonné ; l'ancien mot `Subscribe` survit comme alias de `OnPassed`.

Reaxis a remplacé la contre-pression par le débordement, un mot qu'on n'a pas besoin d'un schéma pour comprendre. Et il a été audité. Une lecture des quatre stratégies que l'article documentait a trouvé que trois d'entre elles perdaient des données sans en signaler aucune. La stratégie par défaut écartait tout ce qui dépassait la capacité pendant que le compteur affichait zéro. C'est corrigé, et prouvé pour les quatre stratégies. Ici, quatre éléments entrent dans un tampon de deux. Que dit le flux de ce qui s'est passé ?

HARO

```
oRs = new stzReactiveSystem()
oS = oRs.CreateStream("probe")
oS {
        SetOverflowStrategy(BUFFER_REJECT_NEWEST, 2)
        for i = 1 to 4
                Receive(i)
        next
        ? OverflowStats()[:currentBuffer]
        #--> 2
        ? OverflowStats()[:droppedCount]
        #--> 2
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il dit que `currentBuffer` vaut 2 et que `droppedCount` vaut 2. Le compteur qui affichait zéro compte désormais !

Et pour ce que le terrain appelle debounce, Reaxis dit ce que cela veut dire : attendre que l'attribut se stabilise. Trois changements en succession rapide : combien de réactions, une fois que la valeur a cessé de changer pendant cinquante millisecondes ?

HARO

```
nSettled = 0
oRs4 = new stzReactive()
oX = oRs4.ReactiveObject()
oX.WaitForAttributeToSettle("query", 50, func(attr, oldV, newV) { nSettled++ })
oX.SetAttribute("query", "s")
oX.SetAttribute("query", "so")
oX.SetAttribute("query", "sof")
oX { ProcessPendingReactions() }
oRs4.RunLoop()
? nSettled
#--> 1
oRs4.Stop()
```

✓ exécuté : a affiché ce que ce bloc promet

Une seule réaction. `nSettled` dit `1`.

Le troisième sujet du chapitre, ce sont les moteurs, les petits paradigmes que la bibliothèque a repensés un par un. Le chapitre 3 vous a donné la vérité et le vide. En voici deux de plus, exécutés ici. Le temps est un espace de travail, pas un nombre. Une ligne de temps tient des plages et calcule les vides et les chevauchements. Que dit-elle du vide entre deux phases, et que fait-elle d'un point qui tomberait dans une région bloquée ?

HARO

```
oTl = new stzTimeLine("2024-01-01 00:00:00", "2024-12-31 23:59:59")
oTl {
        AddSpan("PHASE1", "2024-01-01 00:00:00", "2024-02-15 23:59:59")
        AddSpan("PHASE2", "2024-03-01 00:00:00", "2024-04-15 23:59:59")
        ? len(Gaps())
        #--> 1
        ? Gaps()[1][:after]
        #--> PHASE1
        ? Gaps()[1][:before]
        #--> PHASE2
        ? HasOverlaps()
        #--> FALSE
        AddSpan("PHASE2B", "2024-04-01 00:00:00", "2024-05-15 23:59:59")
        ? HasOverlaps()
        #--> TRUE
}
oB = new stzTimeLine("2024-01-01", "2024-12-31")
oB {
```

```
        AddBlockedSpan("MAINTENANCE", "2024-06-15 09:00:00", "2024-06-15 17:00:00")
        AddPoint("EVENT1", "2024-06-15 08:00:00")
        try
                AddPoint("EVENT2", "2024-06-15 12:00:00")
        catch
                ? cCatchError
        done
        #--> Point 'EVENT2' falls within a blocked span or blocked point
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le vide est nommé par ce qu'il sépare, `PHASE1` et `PHASE2`. La plage bloquée a refusé `EVENT2` ! Le refus porte sa raison : `Point 'EVENT2' falls within a blocked span or blocked point`.

Et une grille a un curseur qui sait où il est : la position est une colonne et une ligne, et se déplacer au-delà d'un bord reste au bord. Deux pas vers le bas et trois vers la droite, puis trois de plus vers la droite : où est le curseur ?

HARO

```
StzGridQ([ 5, 5 ]) {
        MoveDown()
        MoveDown()
        MoveRight()
        MoveRight()
        MoveRight()
        ? @@( CurrentPosition() )
        #--> [ 4, 3 ]
        MoveRight()
        MoveRight()
        MoveRight()
        ? @@( CurrentPosition() )
        #--> [ 5, 3 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

En `[ 4, 3 ]`, puis en `[ 5, 3 ]`. Trois pas de plus vers la droite ont gagné une colonne, et le curseur est resté au bord !

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé de laisser l'expression régulière n'être qu'une façon de lire du texte. J'en ai fait un milieu computationnel. Un déclencheur lance un calcul, le calcul écrit un état quand il change la valeur, et l'état enregistre son pourquoi. Le Regexuter est une faculté et pas un esprit, et il ne tient aucun but. J'ai ajouté le but deux échelons plus haut, au quatrième, dans l'agent, exprès.

J'ai refusé le vocabulaire du terrain pour la programmation réactive. Observables, abonnés, contre-pression, sources chaudes et froides, ordonnanceurs : empruntés à la plomberie, à la téléphonie et à l'édition, et pas au problème. Une fonction en attente est une Rfunction. Les données circulent, et rien ne s'appelle producteur ou consommateur. Le débordement remplace la contre-pression. Attendre qu'un attribut se stabilise remplace le debounce. Les anciennes orthographes marchent encore comme alias : feed, emit, send et receive appellent tous une même méthode, mon changement était donc un changement de nom et pas un retrait.

J'ai refusé de construire les frères. L'article qui raconte le Regexuter en décrit quatre autres : un analyseur, un génétique, un linguistique et un quantique. Dans le génétique les motifs évoluent contre un critère d'adaptation, dans le linguistique les déclencheurs sont des règles de grammaire, et dans le quantique un motif peut être dans plusieurs états à la fois. L'article donne à chacun un exemple d'appel, et aucune classe ne se tient derrière. La conception a trouvé plus tard un but pour deux d'entre eux. Répondre par l'exemple exige l'induction de motifs, le terrain du génétique ; plusieurs réponses acceptables exigent des candidats superposés, réduits par des règles ou par un choix, celui du quantique. Le chapitre 18 dit où cela en est. Ils restent une vision ; deux ont leur but écrit.

J'ai refusé de revendiquer la cascade. L'architecture de la bibliothèque le dit elle-même : l'échafaudage anticipe la cascade mais ne redéclenche pas encore les déclencheurs dépendants jusqu'à un point fixe. Je préfère vous montrer le bloc qui échoue à enchaîner plutôt qu'un bloc qui y arrive.

5. LE TRIBUNAL

Le Regexuter et le Listexuter ont des tests classiques, et aucune garde narrée. Les huit tests classiques du Regexuter ont tourné sans erreur : le journal du dossier dit `PASS 8 ERR 0`. Rien n'y vérifie ce qu'ils impriment. Cinq des sept tests du Listexuter échouent à leur propre sortie attendue, et deux ne promettent rien. La cascade de l'agent est prouvée par `piagent_narrated`, scène cinq. Reaxis a beaucoup de gardes. `00_overflow_narrated` prouve les quatre stratégies de débordement et le cas qui ne perd rien ; `60_reaxis_on_reactor_narrated` prouve un minuteur à coup unique, le minuteur répété, le HTTP non bloquant et l'attribut qui se stabilise sur la boucle du moteur. La ligne de temps a `test_timeline_regression`, qui couvre la construction, la durée, l'ajout de points et le déplacement des bornes, mais ni les vides, ni les chevauchements, ni les plages bloquées. La grille a `00_grid_narrated`, et les opérations profondes `01_deeplist_narrated`.

La section suivante place ce dossier à côté de celui du terrain.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`00_overflow_narrated` 17 assertions
base/test/reactivestream/00_overflow_narrated.ring

`60_reaxis_on_reactor_narrated` 9 assertions
base/test/reactive/60_reaxis_on_reactor_narrated.ring

`piagent_narrated` 14 assertions
base/test/agentic/piagent_narrated.ring

`00_grid_narrated` 24 assertions base/test/grid/00_grid_narrated.ring

`test_timeline_regression` 20 assertions
base/test/timeline/test_timeline_regression.ring

`01_deeplist_narrated` 16 assertions
base/test/list/01_deeplist_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les moteurs de règles du terrain mènent. Ils sont plus anciens et plus mûrs que le Regexuter, ils s'enchaînent jusqu'à un point fixe, et les algorithmes qui les portent ont des décennies. Ce que je revendique n'est pas un moteur de règles. C'est le côté condition : des règles qui sont des phrases des langages de motifs, sur les deux milieux qui ont une machine réagissante construite, le texte pour le Regexuter et les listes pour le Listexuter. L'état garde, pour chaque déclenchement qui change une valeur, son pourquoi. L'architecture même de la bibliothèque appelle le Regexuter un moteur de règles à chaînage avant naissant, et je n'en dirai pas plus.

Les bibliothèques réactives du terrain mènent en portée. Rx existe dans la plupart des langages courants, avec un grand nombre d'opérateurs ; c'est une connaissance générale. Reaxis est une bibliothèque sur une boucle d'événements. Le module propre du moteur pour cela est une petite table de règles sur des sous-chaînes : le modèle Container, Stream et Rfunction vit dans la bibliothèque, sur la boucle d'événements du moteur.

Je n'imprimerai pas la comparaison par étoiles avec les outils du terrain que porte l'article de la bibliothèque sur les objets réactifs. Elle n'est ni datée ni sourcée.

La section suivante dit où tout cela s'arrête.

7. LA LIMITE

CONSTRUIT	le Regexuter et le Listexuter, qui réagissent avec un état qui se souvient ; Reaxis, la programmation réactive en mots simples, sur la boucle d'événements du moteur ; la ligne de temps, la grille et les opérations profondes ; l'agent d'IA dont les compétences s'enchaînent jusqu'à un point fixe
CONÇU	le Softanzuter proprement dit, plusieurs facultés partageant un seul état et s'enchaînant jusqu'à ce qu'il se stabilise ; le moteur universel derrière lui ; ses quatre frères, analyseur, génétique, linguistique et quantique
ABSENT	une chaîne de dépendance qui relie un déclencheur à un autre ; un verbe de fusion de sections que l'article présente comme son point central

Ce qui est construit : le Regexuter et le Listexuter ; Reaxis sur la boucle d'événements du moteur ; la ligne de temps, la grille et les opérations profondes ; l'agent d'IA dont les compétences s'enchaînent.

Ce qui est conçu : le troisième échelon, plusieurs facultés sur un état partagé, s'enchaînant jusqu'à ce qu'il se stabilise. Le moteur universel pour cela est planifié. Les quatre frères sont une vision. Le macroplan marque le module du moteur comme fait, et le tableau d'architecture le marque comme planifié : le même nom pour deux choses. Ce tableau marque comme planifié tout module de sa cinquième couche, y compris ceux qui ont du code source, et il est donc périmé sur ce point. Le milieu universel est planifié dans le document de conception du moteur.

Ce qui est absent ou cassé à ce commit :

- La chaîne de dépendance n'en est pas une. Chaque entrée d'état ne dépend que de son propre déclencheur, de sorte que la chaîne enregistre quel déclencheur était actif et jamais un lien entre deux. Demander la chaîne fait déborder la pile : l'exécution se termine par `Error (R4) : Stack Overflow`. Aucun test ne l'appelle.
- Cinq des sept tests classiques du Listexuter échouent à leur propre sortie attendue, et deux ne promettent rien. Le journal d'erreurs de ce dossier note, dans le sixième fichier, `Error (C8) : Parentheses ')' is missing`.
- Une liste imbriquée qui correspond en entier est signalée deux fois.
- Le minuteur répété de l'article prend trois arguments et la bibliothèque en prend deux ; la forme avec unités est `RunEveryXT`.
- Deux des verbes de chemin de l'article sur les listes profondes, `ItemAtPath` et `ItemsAtPaths`, vivent sur une sous-classe, `stzDeepList`, et pas sur une liste ordinaire ; `Paths()` est sur une liste ordinaire.
- L'article sur la fusion de sections présente comme son point central un verbe général qui n'existe pas. Les deux qui existent, fusionner ce qui se chevauche et fusionner ce qui est inclus, tournent. La fusion de sections n'a pas de fichier de test à elle.
- Il n'y a pas de module natif du moteur pour le modèle Container, Stream et Rfunction : il tourne dans la bibliothèque, et le module réactif propre du moteur est une petite table de règles sur des sous-chaînes. La garde de la ligne de temps ne vérifie ni vide, ni chevauchement, ni plage bloquée ; dans les fichiers classiques de la ligne de temps, sept échouent à leur sortie attendue et dix-huit ne promettent rien. Le navigateur de grille adossé au moteur, `stzGridNav`, n'a aucun test.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Liez un motif à du code, et traitez ; ou déclarez un flux et dites en mots ce qui arrive aux données qui y circulent.
L'analyste	Le temps est un espace de travail : les vides, les chevauchements et les régions bloquées sont calculés, et un point qui tomberait dans une région bloquée est refusé avec la raison.
Le concepteur	L'état d'une réaction qui change une valeur garde son pourquoi : la correspondance, la valeur calculée, la position, le déclencheur.
Le décideur	Le débordement est un nombre que vous pouvez lire : gardé, écarté. Une stratégie qui perdait des données sans le dire a été trouvée par un audit, et corrigée.
Le formateur	L'échelle est un cursus : un motif, une réaction, un état, un but.
L'agent	Votre esprit peut être des compétences qui s'enchaînent sur ce qu'elles perçoivent, jusqu'à ce que le monde cesse de changer.

9. EXERCICE

Écrivez un Regexuter avec deux déclencheurs, un pour les nombres et un pour les nombres de trois chiffres ou plus, avec du code qui transforme un nombre en son triple. Avant de l'exécuter, écrivez sous chaque question ce que vous attendez : quelles sont les correspondances, quels sont les résultats, et si le second déclencheur se déclenche sur le résultat du premier. Exécutez-le. Si le second déclencheur se déclenche, vous avez trouvé quelque chose que je n'ai pas trouvé.

Puis dites en une phrase ce qu'il faudrait changer dans la machine pour qu'il se déclenche, et à quel échelon de l'échelle ce changement appartient.

Je vous laisse celui-ci. Reprenez le deuxième bloc de la section 3 et écrivez ses deux lignes `Trigger` dans l'autre ordre, celui des trois chiffres d'abord :

HARO

```
rxu() {
        Trigger(:Big = "\d{3,}")
        Trigger(:Number = "\d+")
        @C(:When = :Number, :Do = '{ @value = @number(@value) * 100 }')
        @C(:When = :Big, :Do = '{ @value = "BIG" }')
        Process("Found 7 items")
```

```
        ? @@( Matches() )
        ? @@( Results() )
        ? @@( Triggers() )
}
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Avant de l'exécuter, écrivez si l'ordre change ce que contiennent `Results()` et `Triggers()`. Je n'imprime pas la sortie de cette version et je ne vous donne pas la réponse : la section 3 dit contre quoi chaque déclencheur est comparé, et la section 4 dit ce que j'ai refusé d'affirmer. Quand vous saurez dire pourquoi, vous saurez ce qui sépare le deuxième échelon du troisième.

Le chapitre 10 passe de la machine qui pense au code qui change. Son titre est *La programmation devient raffinement*, et il tient à un seul principe : cessez de vouloir gouverner l'intelligence, et gouvernez le changement.

OÙ SONT LES PREUVES

NARRATION	`stzregexuter-regex-as-computational-reactive-medium.md`
NARRATION	`stz-reaxis-reactive-programming-narration.md`
NARRATION	`stz-managing-deep-lists-narration.md`
NARRATION	`softanza-reactive-objects-narration.md`
SUR LE SITE	softanzuter
SUR LE SITE	craft
DÉCISIONS DE L'ÉTUDE	12.4, 12.18, 12.19

PARTIE II · CHAPITRE 10

La programmation devient raffinement

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je dis ce que j'entends par raffiner plutôt que programmer. C'est un changement apporté à du code déclaré modifiable, proposé sous une forme typée, jugé par une barrière, enregistré comme un pas, et réversible. J'exécute la forme que la bibliothèque lui donne : le point de raffinement, la proposition, la barrière, la cascade, le plancher de confiance et le retour en arrière. Et j'imprime la barrière telle qu'elle est, cinq contrôles et non les quatre de ma conception. Je dis où le terrain mène et où la barrière s'arrête. Le principe est la phrase que j'ai écrite sur la quatrième de couverture de mon livre précédent, *Refinement-Oriented Programming* : cessez de vouloir gouverner l'intelligence, et gouvernez le changement.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a un nom pour changer une infrastructure avec une répétition : planifier, puis appliquer. Un outil calcule ce qui changerait, le montre, et ne l'applique que lorsque vous le dites. Terraform a rendu le schéma familier, et les contrôleurs d'admission du terrain appliquent la même idée à une requête avant qu'un cluster l'accepte. Plus récemment, le terrain a un nom pour une façon de travailler avec des agents d'IA qui écrivent du code : les spécifications d'abord, et l'agent remplit. Tout cela est une connaissance générale, pas une étude du dossier.

Ce que ces pratiques partagent, c'est l'endroit où elles se tiennent. Le plan et le contrôle d'admission portent sur l'infrastructure et sur les requêtes. Pour le code lui-même, une bonne part de la réponse du terrain à un changement reste un diff, un correctif de texte, et la relecture d'une personne. Cela marchait tant que chaque changement avait derrière lui une personne à qui poser la question. C'est un instrument faible quand la plupart des changements arrivent d'agents, en volume ; je tiens cela pour une vue, non pour un résultat.

À cet instrument faible, j'oppose une seule phrase.

2. LE PREMIER PRINCIPE

J'ai écrit la phrase sur la quatrième de couverture de mon livre précédent, *Refinement-Oriented Programming*, et je n'en ai pas trouvé de meilleure : **cessez de vouloir gouverner l'intelligence, et gouvernez le changement.** L'en-tête de la bibliothèque dit la même chose sous la forme d'un type : un changement est une proposition typée, pas un diff.

Le reste du chapitre exécute cette phrase.

3. LA FORME SOFTANZA

La pratique a changé de nom dans ma vie. Je disais que je programmais. Je dis maintenant de plus en plus que je raffine, et je veux dire par là quelque chose d'exact. Un raffinement est un changement apporté à du code qui a été déclaré modifiable, proposé sous une forme typée, jugé par une barrière, enregistré comme un pas, et réversible. Mon livre précédent, *Refinement-Oriented Programming*, raconte la discipline à travers l'histoire de Softibank, une banque devant la lettre d'un régulateur. Ce chapitre dit ce que la bibliothèque en porte.

Il commence par déclarer où le code peut changer. Un point de raffinement est un bouton nommé, écrit dans le source sous forme de balise, avec son genre, sa valeur et ses bornes, et le code qui le porte est interrogé dans sa scène, la forme du chapitre 5 :

```
HARO
```

```
cSrc = 'vat_rate = <R:PARAM name="vat" value="0.20" min="0" max="0.25">' + nl +
       'sort_by  = <R:ALGO name="sort" value="quick" options="quick|merge|heap">'
o = new stzRefinableCode(cSrc)
o {
        ? NumberOfPoints()
        #--> 2
        ? @@( RefinementPoints() )
        #--> [ [ "vat", "param", "0.20" ], [ "sort", "algo", "quick" ] ]
        ? KindOf("vat")
        #--> param
}
```

✓ exécuté : a affiché ce que ce bloc promet

Un changement est alors une proposition. Elle nomme un point et une valeur ; si elle franchit la barrière, elle prend effet et est enregistrée comme un seul pas :

HARO

```
o {
        r = Refine("vat").To("0.22")
        ? r[:admitted]
        #--> 1
        ? ValueOf("vat")
        #--> 0.22
        ? NumberOfSteps()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'admission s'imprime en `1` et `0`, non en `TRUE` et `FALSE` : c'est ainsi que ce verbe répond, et j'imprime l'exécution. Et que change une proposition qui ne franchit pas la barrière, et que dit la barrière ? Exécutez-le :

HARO

```
o {
        r2 = Refine("vat").To("0.99")
        ? r2[:admitted]
        #--> 0
        ? r2[:why]
        #--> constraint: 0.99 above max 0.25
        ? ValueOf("vat")
        #--> 0.22
```

```
        ? Refine("sort").To("bubble")[:why]
        #--> constraint: 'bubble' not in options quick|merge|heap
        ? Refine("ghost").To("1")[:why]
        #--> structural: no refinement point 'ghost'
        ? NumberOfSteps()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Trois refus, et la ligne de temps n'a pas grandi d'un pas ! Le commentaire de la bibliothèque en fait une de ses lois : une proposition rejetée ne finit jamais modifiée. Chaque refus nomme d'abord son étape, `constraint` deux fois et `structural` une fois, comme dans `structural: no refinement point 'ghost'`, pour qu'une personne, ou un agent, puisse corriger la proposition sans lire un diff.

La cascade est ce que vous lisez à la place du diff. Avant le changement, demandez ce qu'il atteint. Il y a deux cascades, et elles sont séparées. Pour un point, c'est la ligne où il se trouve et les autres points de cette même ligne :

HARO

```
cPrice = 'price = <R:PARAM name="vat" value="0.20" min="0" max="0.30"> * <R:PARAM name="margin"
value="1.1" min="1" max="2">' + nl +
         'floor = <R:PARAM name="floor" value="0.10" min="0" max="0.30">'
oc = new stzRefinableCode(cPrice)
oc {
        aC = Cascade("vat")
        ? aC[:lines][1]
        #--> 1
        ? @@( aC[:alsoTouches] )
        #--> [ "margin" ]
        ? Cascade("floor")[:lines][1]
        #--> 2
        ? len(Cascade("floor")[:alsoTouches])
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

`vat` se trouve ligne 1 et touche aussi `[ "margin" ]` ; `floor` se trouve ligne 2, seul.

Pour une classe, la cascade est une propriété de la structure du code : qui en descend et qui hérite d'une méthode. La bibliothèque la calcule à partir du graphe de code, et elle peut analyser un texte sans toucher un fichier :

HARO

```
cCls = 'class Tariff' + nl +
       '        def Rate()' + nl +
       '                return 0.20' + nl +
       'class Invoice from Tariff' + nl +
       '        def Total()' + nl +
       '                return 1' + nl +
       'class Receipt from Invoice' + nl +
       '        def Print()' + nl +
       '                return 1' + nl +
       'class Quote from Tariff' + nl +
       '        def Rate()' + nl +
       '                return 0.18' + nl
g0 = new stzRingCodeGraph("")
g0 {
        ScanSource(cCls, "world.ring")
        BuildGraph()
        ? @@( Cascade("Tariff")[:descendants] )
        #--> [ "Invoice", "Quote", "Receipt" ]
        ? Cascade("Tariff")[:blastRadius]
        #--> 3
        ? @@( ImpactOf("Rate")[:inheritedBy] )
        #--> [ "Invoice", "Receipt" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Changez le taux du tarif et trois classes sont dans le rayon d'impact. Deux d'entre elles héritent de la méthode, et `ImpactOf` les nomme : `[ "Invoice", "Receipt" ]`. La troisième, `Quote`, la redéfinit, de sorte qu'elle la possède et ne fait pas que l'hériter. C'est la différence que trace `ImpactOf`.

Comment un changement atteint-il ses dépendants ? Par une règle qui couvre plusieurs points, jugée sur l'état que le changement produirait. Si la règle échoue, la valeur est remise :

HARO

```
d = new stzRefinableCode(cPrice)
d {
```

```
        DeclareDerivation("vat-above-floor",
                func oCode { return ring_number(oCode.ValueOf("vat")) >=
ring_number(oCode.ValueOf("floor")) },
                "vat must not drop below floor")
        ? Refine("vat").To("0.30")[:admitted]
        #--> 1
        r3 = Refine("vat").To("0.05")
        ? r3[:admitted]
        #--> 0
        ? r3[:why]
        #--> derivation: rule 'vat-above-floor' violated -- vat must not drop below floor
        ? ValueOf("vat")
        #--> 0.30
}
```

✓ exécuté : a affiché ce que ce bloc promet

La règle a refusé le changement et s'est nommée : `derivation: rule 'vat-above-floor' violated -- vat must not drop below floor`. La valeur est toujours `0.30`.

Et qui l'a proposé ? Chaque proposition porte son origine, et un point peut fixer un plancher de confiance qui refuse une origine inférieure, sans refuser une personne. Qu'arrive-t-il à une proposition venue d'un modèle de langage quand le plancher du point exige une origine de confiance ? Exécutez-le :

HARO

```
d {
        TrustFloor("vat", :trusted)
        r4 = Refine("vat").As(:llm).To("0.25")
        ? r4[:admitted]
        #--> 0
        ? r4[:why]
        #--> trust: posture :llm below the required floor for 'vat'
        ? ValueOf("vat")
        #--> 0.30
        ? Refine("vat").As(:trusted).To("0.25")[:admitted]
        #--> 1
        ? PostureOf(NumberOfSteps())
        #--> trusted
}
```

✓ exécuté : a affiché ce que ce bloc promet

La proposition du modèle a été refusée : `trust: posture :llm below the required floor for 'vat'`. Le point a gardé `0.30` ! La même proposition, venue d'une origine de confiance, a été admise, et le pas enregistre `trusted`.

Et qui peut tourner quel bouton ? C'est déclaré lui aussi : une permission, et un niveau d'autorité contre un niveau de risque. Prenez un acteur délégué et un bouton du troisième niveau. Exécutez-le :

HARO

```
gv = new stzRefinableCode(cPrice)
gv {
        SetGovernedBy(new stzGovernance("release-ops"))
        SetActor("release-bot")
        SetRiskFor("vat", 3)
        SetAllowRefine("vat")
        SetAuthorityLevel(:Delegated)
        r5 = Refine("vat").To("0.22")
        ? r5[:admitted]
        #--> 0
        ? r5[:why]
        #--> governance: refused: 'release-bot' holds 'delegated' authority (level 2) but
'refine-vat' is risk tier 3 (SHOULD does not cover it)
        ? ValueOf("vat")
        #--> 0.20
        SetAuthorityLevel(:Autonomous)
        ? Refine("vat").To("0.22")[:admitted]
        #--> 1
        ? GovernanceQ().NumberOfDecisions()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'acteur délégué a été refusé, `'release-bot' holds 'delegated' authority (level 2)`, et la valeur est restée `0.20`. La même proposition a été admise une fois l'acteur escaladé ! La décision est enregistrée : `NumberOfDecisions()` a imprimé `1`.

Enfin, un pas peut-il être repris ? La réversibilité fait partie du modèle de données : chaque pas admis garde sa valeur antérieure. Remontez jusqu'à un point de contrôle nommé, remontez jusqu'à zéro, avancez de nouveau :

HARO

```
t = new stzRefinableCode(cPrice)
t {
        Refine("vat").To("0.21")
        Refine("vat").To("0.22")
        Checkpoint("reviewed")
        Refine("vat").To("0.28")
        ? ValueOf("vat")
        #--> 0.28
        RevertToCheckpoint("reviewed")
        ? ValueOf("vat")
        #--> 0.22
        ? NumberOfSteps()
        #--> 2
        RevertTo(0)
        ? ValueOf("vat")
        #--> 0.20
        ? CanRevert()
        #--> 0
        Redo()
        ? ValueOf("vat")
        #--> 0.21
}
```

✓ exécuté : a affiché ce que ce bloc promet

À zéro, `CanRevert()` a imprimé `0`. Un seul `Redo()`, et la valeur est `0.21`, le premier pas de nouveau.

Voilà la barrière sous la forme que la bibliothèque lui donne. Cinq contrôles, dans l'ordre où le code les exécute. La structure de la proposition, puis les contraintes du point, puis la gouvernance, puis la confiance de l'origine, et enfin la dérivation entre les points. Je les ai exécutés plus haut dans l'ordre du récit, non dans celui du code. L'en-tête du fichier appelle cela une barrière à quatre étapes sur une ligne et à cinq sur une autre. Le code numérote ses contrôles 1, 2, 3, 4 et 4b. Le contrôle de confiance a été ajouté aux quatre le 16 juillet 2026, et le compte de la conception n'a pas été changé. Je rapporte ce qui tourne : cinq contrôles.

Derrière la barrière se tiennent des décisions, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé de gouverner l'intelligence. Une bonne part de l'approche du terrain pour les agents d'IA est de rendre le modèle plus sûr : alignement, garde-fous, une consigne qui demande confirmation ; cela aussi est une connaissance générale. J'ai refusé ce cadre. La sortie d'un modèle est ce qu'elle est, de quelque modèle que ce soit, et l'intelligence de la génération suivante ne restera pas immobile pour que je la gouverne. Le changement est ce qui reste, et c'est lui qui atteint le monde. La barrière se tient donc là où le changement arrive, et elle ne demande qu'une chose : qu'est-ce qui est proposé, d'où, et le code dit-il que ce point peut changer ainsi.

J'ai refusé le diff comme unité. Un diff est un changement de texte. Un raffinement est une proposition typée sur un point déclaré, avec un genre, des bornes et une origine. J'ai gardé le diff comme artefact, et je l'ai refusé comme ce qu'il faut relire.

J'ai fait porter le verbe à la bibliothèque, et laissé l'environnement ailleurs. Le raffinement comme verbe de bibliothèque, le point, la barrière, la cascade et le retour en arrière, appartient au socle commun ouvert. L'environnement dans lequel une équipe vit avec lui au jour le jour, les écrans, le mode réel, le livre, est un produit. Il ne déborde pas dans la bibliothèque.

J'ai imprimé la barrière telle qu'elle est. J'avais écrit la conception avec quatre étapes, et le code a cinq contrôles. Une page qui citerait ma conception et pas mon code aurait tort, et c'est le code qui tourne.

5. LE TRIBUNAL

`refinablecode_narrated` prouve le point déclaré, la proposition qui passe, les deux refus qui laissent le source entier (bornes et options), la cascade, le retour en arrière et le fichier sauvegardé. `refine_deepening_narrated` prouve la ligne de temps avec ses points de contrôle et son rétablissement, et le plancher de confiance. `gate_deepening_narrated` prouve le refus d'un point inconnu, la dérivation entre les points avec son annulation, et le refus gouverné avec sa filiation.

Vient ensuite le terrain, et où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`refinablecode_narrated` 17 assertions
base/test/refine/refinablecode_narrated.ring

`refine_deepening_narrated` 24 assertions
base/test/refine/refine_deepening_narrated.ring

`gate_deepening_narrated` 19 assertions
base/test/refine/gate_deepening_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en infrastructure, où planifier et appliquer sont la pratique quotidienne de nombreuses équipes et où l'application parle à de vrais fournisseurs ; c'est une connaissance générale, comme dans la section 1. La barrière de Softanza porte sur le changement apporté au code, que le terrain traite par une relecture.

Les outils du terrain mènent en étendue : ils ne sont liés ni à une bibliothèque, ni à un langage, ni à un bouton déclaré. La barrière de Softanza juge ce que le code a déclaré comme point. Un changement apporté à un endroit qui n'a jamais été

déclaré n'est pas un raffinement ici ; il est hors de la barrière.

Et sur la confiance, une bonne part de l'approche du terrain est faite pour passer à l'échelle de n'importe quel modèle d'IA. La mienne repose sur une déclaration par point.

Voyons maintenant où la barrière s'arrête.

7. LA LIMITE

CONSTRUIT — le point de raffinement déclaré dans le source ; la barrière à cinq contrôles ; la cascade d'un point et d'une classe ; le retour en arrière par pas, par point de contrôle et jusqu'à zéro ; le plancher de confiance et le raffinement gouverné

CONÇU — l'expérience que décrit l'article : demander à un point de s'expliquer, appliquer un plan, un raffinement en une ligne ; la cascade du graphe de code jointe à la barrière ; le siège de raffinement de Haro

ABSENT — un historique sauvegardé : le fichier garde le source et rien de la ligne de temps

Ce qui est construit : le point de raffinement, la barrière avec ses cinq contrôles, la cascade d'un point et d'une classe, le retour en arrière, le plancher de confiance et le raffinement gouverné. Les gardes de la section 5 les couvrent, avec deux exceptions. La cascade d'une classe n'a pas de garde ici, parce que la garde du graphe de code scanne un chemin vivant de l'arbre de travail de la bibliothèque. Et aucune garde n'affirme les autres points d'une ligne.

Ce qui est conçu, et absent de la bibliothèque par son nom : l'expérience que décrit l'article d'origine. La classe que cet article nomme, `stzPolyCode`, n'est pas dans la bibliothèque à ce commit. Le raffinement en une ligne du document de conception, un point qui s'explique, un plan qui est appliqué, ne sont pas dans la bibliothèque. La cascade du graphe de code et la cascade de la barrière ne sont pas jointes. Et Haro a une barrière dans un spike écrit uniquement dans l'ancien langage hôte, son propre siège de raffinement n'étant pas construit.

Ce qui est construit, et où cela s'arrête : les limites de la barrière telle qu'elle est.

- Elle vérifie les contraintes de trois genres de point, un paramètre, un algorithme et une bibliothèque. Un point d'un autre genre, un bloc de code par exemple, est analysé et passe sans contrôle ; les contrôles pour ces genres sont l'étape suivante.
- Une posture de confiance inconnue se classe comme de confiance : le code lui donne le rang le plus haut. Une proposition venue d'une origine que le code ne connaît pas franchit donc un plancher de confiance.
- Une dérivation est appliquée à titre provisoire et restaurée en cas d'échec. Je dis donc « ne finit jamais modifié », pas « jamais touché ».
- Le fichier sauvegardé contient le source rendu ; l'historique, les postures, la ligne de temps et le rechargement sont absents, et c'est l'étape suivante. La réversibilité vit dans l'objet en cours d'exécution ; la porter dans le fichier est encore à venir. La bibliothèque écrit ce fichier avec le suffixe `.zrfn`, tel que livré. La charte de Haro, ratifiée le 1er octobre 2026, retient le suffixe `.rfn` pour la phrase de raffinement de Haro, et laisse `.zrfn` à l'extraction de la bibliothèque par balises de commentaire, la forme que ce chapitre a exécutée.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Déclarez les endroits qui peuvent changer, et proposez-leur des changements sous forme de propositions typées. Un refus nomme son étape.
L'analyste	La règle qui couvre plusieurs points, c'est à vous de la déclarer, et la barrière la juge sur l'état que le changement produirait.
Le concepteur	La cascade est ce que vous lisez à la place du diff : pour un point, quelle ligne et quels points sur elle ; pour une classe, quelles classes héritent. Ce sont deux cascades, pas encore une seule.
Le décideur	Qui peut tourner quel bouton est une permission et une autorité contre un niveau de risque, et la décision est enregistrée avec son autorité du moment.
Le formateur	Reculez d'un pas, remontez jusqu'à un point de contrôle nommé, remontez jusqu'à zéro, avancez de nouveau. Un apprenant peut essayer sans crainte parce que chaque pas garde sa valeur antérieure.
L'agent	Vous pouvez proposer. Votre origine est sur la proposition, et un point peut fixer un plancher que votre origine ne franchit pas.

9. EXERCICE

Prenez deux réglages d'un de vos programmes : un taux et un plafond, par exemple, avec une règle selon laquelle le taux doit rester sous le plafond. Déclarez-les comme des points, avec des bornes, et déclarez la règle qui les couvre. Avant de rien exécuter, écrivez sous chaque proposition si elle est admise et, sinon, à quelle étape elle est refusée. Proposez un changement du taux au-dessus du plafond, et un autre depuis une origine que vous déclarez comme un modèle d'IA, contre un plancher que vous fixez. Puis remontez jusqu'à zéro et avancez d'un pas.

Exécutez-le. Si vous vous êtes trompé sur l'étape, vous avez trouvé un endroit où votre propre barrière est en désaccord avec votre propre attente.

Je vous laisse celui-ci. La section 7 dit ce que la barrière fait d'une origine que le code ne connaît pas, et le bloc de confiance de la section 3 est celui à exécuter. Ses premières lignes sont ci-dessous, et `d` est l'objet de ce bloc, alors exécutez d'abord la

section 3 ; changez une seule chose, et remplacez `:llm` par un nom de votre choix, que le code n'a jamais appris.

HARO

```
d.TrustFloor("vat", :trusted)
r4 = d.Refine("vat").As(:yourorigin).To("0.25")
? r4[:admitted]
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le, et lisez ce que fait la barrière en regard de ce que dit la section 7. Je ne vous dirai pas si la barrière a raison. Quand vous aurez tranché, vous saurez ce que le plancher de confiance fait d'une origine qu'il n'a jamais rencontrée.

Le chapitre suivant énonce une loi : un langage est déclaré et fermé, et ce qui le parle, une personne ou un modèle, est jugé par lui. Ici, la chose jugée était un changement ; là, c'est ce qui parle.

OÙ SONT LES PREUVES

NARRATION	`stzpolycode-refinement-centered_programming.md`
SUR LE SITE	refinement
DÉCISIONS DE L'ÉTUDE	12.19, 12.23, 12.29

PARTIE III

Les langages, repensés

> *Un langage est déclaré et fermé, et tout ce qui le parle, personne ou modèle, est jugé par lui.*

11 · Loi : un langage est déclaré et fermé, et ce qui le parle, personne ou modèle, est jugé par lui
12 · Un langage, déclaré
13 · Naturel, et exécutable : deux directions, une seule discipline
14 · La programmation de la connaissance
15 · Les langages de motifs
16 · Sept langages, une porte

PARTIE III · CHAPITRE 11 · COLONNE

Loi : un langage est déclaré et fermé, et ce qui le parle, personne ou modèle, est jugé par lui

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:58 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 11 bloc(s) sur 11 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, j'énonce la loi de la partie III et je la fais tourner. Je commence par une phrase qui tourne parce que chaque mot est dans un dictionnaire que la bibliothèque possède, en anglais, en hausa, en français et en arabe. Je montre ce qu'elle dit d'un mot qu'elle ne connaît pas. Puis je déclare la forme d'une réponse comme un schéma, je juge des réponses contre lui, et je montre la seule chose qu'une grammaire ne peut pas tuer : une réponse fausse dans la bonne forme. Je dis quelles gardes le prouvent, où le terrain est en avance, et où la couche naturelle n'est pas encore fermée, ce que la loi énonce. La loi elle-même tient en une phrase : un langage est déclaré et fermé, et ce qui le parle, une personne ou un modèle, est jugé par lui.

CE QUE L'ÂGE DE L'IA CHANGE ICI

Dans la partie II, l'âge a changé qui écrit le code. Ici, il change qui prononce la phrase. Une phrase d'un langage avait autrefois une personne derrière elle, quelqu'un à qui l'on pouvait demander ce qu'il voulait dire. Aujourd'hui l'auteur d'une phrase peut être un modèle d'IA, à qui l'on ne peut rien demander, et qui écrira une phrase fluide qu'elle soit vraie ou non.

La grammaire doit donc être fermée, et le tribunal, la partie qui accepte ou refuse une phrase selon sa grammaire, doit la lire. Un langage dont les phrases peuvent contenir n'importe quoi ne peut rien refuser, et un tribunal ne peut pas juger une phrase qu'il ne peut pas lire. La vision de la plateforme met tout le mouvement en une ligne, que j'ai ratifiée : les humains et les machines se rencontrent dans des langages déclarés, petits, fermés et jugés, pas dans du code généraliste.

C'est aussi là que l'ordre du livre se retourne. La couverture va du familier au nouveau : la programmation, les langages, l'intelligence. La logique du dossier va dans l'autre sens. Le premier acte est de déclarer un langage. La programmation, la discipline de la partie II, se passe à l'intérieur de langages déclarés : les formes du vocabulaire sont déjà une déclaration en Haro, dans un spike, une courte expérience écrite pour essayer une conception, hors de la bibliothèque. L'intelligence, partie IV, est gouvernée par les deux : un modèle est tenu à une grammaire déclarée avant de parler, et sa phrase est jugée après. Le livre garde l'ordre de la couverture, donc ce chapitre est au milieu ; dans l'ordre du dossier, il viendrait en premier.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Pour un programmeur qui travaille, un langage est un hôte généraliste, Python, JavaScript, Java. Un « langage de domaine » est d'ordinaire une bibliothèque dont les appels se lisent comme des phrases, alors que le fichier autour peut contenir n'importe quel code de l'hôte. L'étude que le domaine de projets a faite du terrain, écrite en août 2026, nomme ce qui est arrivé à ces conceptions. Les langages internes sont morts d'ouverture, parce qu'un fichier de construction n'est déclaratif que jusqu'au jour où quelqu'un y écrit du code arbitraire. Les ateliers de langage sont morts du poids de leurs outils. Et elle nomme les langages qui ont gagné : SQL, les expressions régulières, Make, HTML et CSS, les langages de configuration. Son verdict est qu'ils ont gagné parce qu'ils sont fermés.

L'ère des modèles a ajouté une pratique : quand un modèle d'IA écrit, ou bien l'on vérifie ce qu'il a écrit, ou bien l'on contraint sa sortie pendant qu'il écrit, avec une grammaire ou un schéma. L'implémentation de référence de l'échantillonnage contraint par une grammaire est un projet ouvert, et le dossier le dit là où il emploie

le même algorithme. Et dans le langage commun du terrain aujourd'hui, « programmer en langage naturel » veut dire une seule chose : donner une consigne à un modèle.

2. LE PREMIER PRINCIPE

Un langage est déclaré et fermé, et ce qui le parle, une personne ou un modèle, est jugé par lui.

3. LA FORME SOFTANZA

Une phrase peut-elle tourner sans qu'un modèle d'IA devine ce qu'elle veut dire ? Elle le peut, quand chaque mot est dans un dictionnaire que la bibliothèque possède :

HARO

```
Naturally("
    Make a string with 'hello niger' inside
    Spacify it and uppercase it
    @Box it while the box@ is rounded
    Display it on the screen
")
#--> ╭─────────────────╮
#    │ H E L L O  N I G E R │
#    ╰─────────────────╯
```

✓ exécuté : a affiché ce que ce bloc promet

Quatre lignes d'anglais sont devenues une boîte arrondie autour de majuscules espacées.

Le dictionnaire est fait de données, indexées par des sens, pas par des mots anglais. Le même sens peut-il donc être dit dans une autre langue en ajoutant un bloc de données, pas de code ? Voici une même instruction en anglais, en hausa, en français et en arabe. La phrase en hausa est celle de la narration, et celle en français vient de la garde de la bibliothèque avec ses données changées. J'ai composé celle en arabe à partir du vocabulaire de la bibliothèque. Aucun locuteur natif n'a lu le hausa ni l'arabe pour autant que le dossier le montre :

HARO

```
? Naturally("Create a string with 'hello' and uppercase it").Result()
#--> HELLO
? NaturallyIn("hausa", "Yi rubutu dauke 'hello' Maida shi").Result()
#--> HELLO
? NaturallyIn("fr", "Crée une chaine avec 'hello' et mets la en majuscules").Result()
#--> HELLO
? NaturallyIn("ar", "أنشئ نصا بـــ 'hello' ثم كبّره").Result()
#--> HELLO
```

✓ exécuté : a affiché ce que ce bloc promet

Quatre langues, un seul sens, et `HELLO` quatre fois.

Et un programme redit ce qu'il a compris, dans sa propre langue. Une personne qui lit l'arabe peut vérifier la lecture que la machine fait de sa phrase sans lire une ligne de code.

HARO

```
o1 = NaturallyIn("ar", "أنشئ قائمة بـــ [ 5, 3, 5 ] ثم أزل تكراراتها")
? @@( o1.Result() )
#--> [ 5, 3 ]
? o1.Understood()
#--> أنشئ قائمة بـــ [ 5, 3, 5 ] -> أزل التكرارات
```

✓ exécuté : a affiché ce que ce bloc promet

Le programme a redit `أنشئ قائمة بـــ [ 5, 3, 5 ] -> أزل التكرارات` : la liste reçue, et la suppression des doublons.

Voici maintenant ce que la bibliothèque fait par règle : dire ce qu'elle n'a pas compris. Que fait-elle d'un verbe qu'elle ne connaît pas ? Écrivez-en un de travers et exécutez :

HARO

```
o1 = Naturally("Create a string with 'softanza' Uppercse it")
? o1.Result()
#--> softanza
? @@( o1.Unresolved() )
#--> [ [ "Uppercse", "uppercase" ] ]
```

```
? o1.UnderstoodAll()
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Le verbe mal orthographié n'est pas deviné. `Unresolved()` répond `[ [ "Uppercse", "uppercase" ] ]`, le mot rencontré à côté du mot que la bibliothèque pense que vous vouliez, et `UnderstoodAll()` répond `0` : le programme admet qu'il n'a pas tout compris. C'est le mode permissif : un mot inconnu est sauté et signalé. Un agent d'IA reçoit le mode strict, qui refuse, et un monde, une liste déclarée d'opérations permises, peut limiter ses phrases à cette liste :

HARO

```
try
        NaturallyStrict("Create a string with 'x' Fooify it")
catch
        ? cCatchError
done
#--> Strict natural mode: not understood: 'Fooify' (did you mean 'boxify'?)
try
        NaturallyStrictIn("en", "Create a list with [ 3, 1 ] Reverse it", [
"METHOD_REMOVEDUPLICATES" ])
catch
        ? cCatchError
done
#--> Operation not permitted in this world: METHOD_REVERSE
```

✓ exécuté : a affiché ce que ce bloc promet

Le mode strict refuse `'Fooify'` et nomme tout de même le mot le plus proche : `did you mean 'boxify'?`. Dans ce monde, une phrase qui renverse une liste est refusée ! Le refus nomme l'opération : `Operation not permitted in this world: METHOD_REVERSE`.

L'autre direction du code naturel dans Softanza est la chaîne quasi naturelle, où la personne est l'analyseur et où la chaîne se lit comme une phrase. Quand elle répond non, dit-elle pourquoi ?

HARO

```
? IsQ().TheQ().LengthQ().OfQ("softanza").LessThanQ().TheQ().LengthQ().Of("Haro")
#--> 0
? Why()
#--> no: 8 is not less than 4
```

✓ exécuté : a affiché ce que ce bloc promet

La chaîne a répondu `0`, et `Why()` a donné la raison : `no: 8 is not less than 4`. Un verdict dit pourquoi.

Voici maintenant la loi à son plus tranchant : un modèle d'IA. La bibliothèque déclare la forme d'une réponse comme un schéma. Que devient un schéma qui nomme un type que la bibliothèque n'a pas ? Il est jugé quand il est déclaré, avant qu'aucun modèle soit appelé :

HARO

```
try
        StzOutputSchemaQ([ [ :field = "x", :type = :colour ] ])
catch
        ? cCatchError
done
#--> stzOutputSchema: unknown field type 'colour' (use
string|number|boolean|oneof|list|structure).
```

✓ exécuté : a affiché ce que ce bloc promet

Le refus nomme le type inconnu, `'colour'`, et liste les six qu'il accepte : `(use string|number|boolean|oneof|list|structure)`. Le schéma a été refusé avant qu'aucun modèle soit appelé ! Un schéma correct est un petit langage fermé. Le tribunal juge n'importe quelle réponse contre lui, quel qu'en soit l'auteur. Et le schéma émet sa grammaire, pour qu'un modèle puisse y être tenu pendant qu'il écrit :

HARO

```
StzOutputSchemaQ([
        [ :field = "name", :type = :string ],
        [ :field = "age",  :type = :number, :must = [ [ ">=", 0 ], [ "<=", 130 ] ] ],
        [ :field = "mood", :type = :oneof,  :choices = [ "positive", "negative" ] ]
]) {
```

```
        ? Accepts("name: Hadiza" + nl + "age: 41" + nl + "mood: positive")
        #--> 1
        ? Accepts("name: Hadiza" + nl + "age: 41" + nl + "mood: angry")
        #--> 0
        ? Accepts("name: Hadiza" + nl + "age: 900000" + nl + "mood: positive")
        #--> 0
        ? Accepts("name: Hadiza" + nl + "age: 62" + nl + "mood: positive")
        #--> 1
        ? ToGBNF()
        #--> root ::= name-line age-line mood-line
        #    name-line ::= "name: " text "\n"
        #    age-line ::= "age: " number "\n"
        #    mood-line ::= "mood: " ("positive" | "negative") "\n"
        #    text ::= [^\n]+
        #    number ::= "-"? [0-9]+ ("." [0-9]+)?
        #    boolean ::= ("yes" | "no" | "true" | "false")
}
```

✓ exécuté : a affiché ce que ce bloc promet

Regardez les quatre verdicts. La première réponse a imprimé `1`. La deuxième, avec `mood: angry`, et la troisième, avec `age: 900000`, ont imprimé `0` : le tribunal les a refusées. La quatrième a imprimé `1` de nouveau. Hadiza a quarante et un ans dans la première réponse et soixante-deux dans la quatrième, et le tribunal accepte les deux ! Une grammaire tue les phrases mal formées, pas les phrases fausses. Le dossier le dit en cinq mots (dans l'original anglais) : structure tue l'informe, pas le faux. Ce livre dit la même chose : la vérité d'une réponse n'est pas l'affaire de la grammaire. Le dossier dit que la vérité demande la réversibilité, un auditeur et un Principal, la personne qui tranche. Une grammaire est la barrière, pas le juge de tout. La grammaire imprime aussi une règle `boolean` qu'aucun champ de ce schéma n'emploie.

Un gabarit est aussi un contrat. Une phrase avec un trou non rempli est refusée avant de tourner :

HARO

```
try
        NaturallyWith("Create a string with {#1} and {#2} it", [ "x" ])
catch
        ? cCatchError
done
#--> Unfilled template hole! The narration still contains a {#n} slot with no value for it.
```

✓ exécuté : a affiché ce que ce bloc promet

Et les motifs sont de petits langages à part entière, mais pas encore fermés : le chapitre 15 montre les classes de motifs acceptant sans se plaindre un mot qu'elles ne connaissent pas. La forme d'une liste est déclarée, pas écrite comme une boucle :

HARO

```
? Lx("[@N, @S]").Match([42, "hello"])
#--> 1
? Lx("[@N, @S]").Match(["hello", 42])
#--> 0
? Lx("[@N, [@N2], @N]").Match([1, [2, 3], 4])
#--> 1
? Lx("[@N, [@N2], @N]").Match([1, [2], 4])
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Deux formes ont correspondu, deux non, et aucune boucle n'a été écrite.

Les chapitres de cette partie prouvent la loi de cinq côtés. Le chapitre 12 raconte un langage déclaré : la forge (la machinerie qui produit l'analyseur, le tribunal, l'imprimeur et la grammaire d'un langage à partir de sa seule déclaration), la grammaire fermée sans échappée vers l'hôte, et le tribunal né avec la grammaire. Il raconte aussi la grammaire émise pour qu'un modèle ne puisse dire que des phrases valides, et les parties qu'une déclaration produit. Le chapitre 13 raconte les deux directions du code naturel, la machine comme analyseur et la personne comme analyseur, dans une seule discipline. Le chapitre 14 raconte la programmation par la connaissance, un fichier qui est un système. Le chapitre 15 raconte les langages de motifs, l'expression régulière élevée aux listes, aux nombres, aux tables, aux matrices, au temps et aux graphes. Le chapitre 16 raconte les sept langages que Softanza atteint par une seule porte gouvernée, et pourquoi une porte n'est pas une facette.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai choisi l'acte : déclarer un langage. Pas « utilisez notre langage ». Déclarez le vôtre : le langage de votre domaine, dans vos mots. C'est le noyau de la plateforme, et c'est la raison d'être de cette partie.

J'ai refusé l'échappée ouverte. Le méta-langage qui déclare les langages du domaine de projets vit hors de cette bibliothèque. Sa règle fondatrice refuse de définir toute forme qui laisserait une phrase atteindre le code de l'hôte, appeler une fonction, évaluer une chaîne ou toucher un fichier. Pas interdit : inexprimable. Un langage interne qui ne peut pas refuser le code arbitraire de l'hôte est, selon les mots de l'étude de terrain du domaine de projets, une API déguisée en langage.

J'ai refusé le langage naturel comme façon de déclarer. Le langage naturel demande ; il ne déclare jamais. Un modèle peut rédiger une déclaration, et, par la règle de la vision, qui est conçue et pas encore construite, le brouillon n'entre dans le monde que comme une proposition : le modèle propose, le tribunal décide, une personne ratifie.

J'ai tenu la promesse de 2020 et changé son mécanisme. Ma première diapositive promettait un langage naturel exécutable. La vision de 2026 a jugé ce slogan : il promettait plus qu'un tribunal ne peut défendre, parce qu'une phrase naturelle peut être ambiguë et qu'un mensonge peut être grammatical. Le naturel reste donc à la porte, là où une phrase demande ou rédige, et l'exécution appartient à des langages fermés qu'un tribunal peut juger. J'ai construit la couche naturelle ainsi bien avant que la vision le nomme. Seuls les mots définis dans le dictionnaire sont interprétés. Tout autre mot est sauté et signalé, ou refusé en mode strict, jamais deviné.

J'ai sorti la gouvernance de la bibliothèque et gardé les refus dedans. En juillet 2026, un kit pour gouverner les agents d'IA a atterri dans la couche naturelle. Je l'ai retiré le jour même : le mode strict, les refus, le pourquoi et le compris appartiennent à la bibliothèque ; un produit pour gouverner les agents, non. Et j'ai

retiré un ensemble de marqueurs, des suffixes qui marquaient une référence, que le chapitre 13 raconte : leur chiffre de l'ordre des lettres avait été inversé par son propre inventeur, de mémoire, et ils répétaient des sens. Un langage qu'une personne parle ne marque jamais une référence de cette façon. Une chaîne non grammaticale qui tourne est un bogue.

J'ai refusé un second plongement. Le chapitre 1 énonce cette décision ; j'ajoute seulement ici qu'une porte apporte un runtime étranger, qu'une facette apporte une syntaxe familière sur la même grammaire, et que le chapitre 16 raconte la différence.

5. LE TRIBUNAL

`multilingual_pack_narrated` prouve les langues comme données : les paquets, les programmes en français et en arabe, un paquet ajouté à l'exécution, et le programme qui se paraphrase dans sa propre langue. `semantic_resolver_narrated` prouve le mot inconnu avec sa suggestion, le mode strict, et la liste d'autorisation sous laquelle tout ce qui est hors liste est grammaticalement impossible. Son exécution n'atteint pas encore sa fin à ce commit. Je l'ai exécutée deux fois le 8 octobre 2026, pour le chapitre 13, et chaque fois elle s'est arrêtée sur une erreur levée par une fonction de découpage de chaîne, avant sa dernière assertion. La page compte donc ses assertions et ne revendique pas son verdict. `literate_templates_narrated` prouve qu'un gabarit est un contrat. `question_narrated` prouve qu'un verdict par oui ou non enregistre son pourquoi. `structuredoutput_narrated` prouve le tribunal de la sortie structurée, y compris le mensonge qu'il laisse passer. `schema_grammar_narrated` prouve le compilateur de grammaire, et `constrained_decoding_narrated` l'échantillonneur qui y tient un modèle.

Sept gardes prouvent la loi ; la section suivante dit où le terrain est en avance.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`multilingual_pack_narrated` 45 assertions
base/test/natural/multilingual_pack_narrated.ring

`semantic_resolver_narrated` 83 assertions
base/test/natural/semantic_resolver_narrated.ring

`literate_templates_narrated` 10 assertions
base/test/natural/literate_templates_narrated.ring

`question_narrated` 21 assertions
base/test/natural/question_narrated.ring

`structuredoutput_narrated` 73 assertions
base/test/neural/structuredoutput_narrated.ring

`schema_grammar_narrated` 28 assertions
base/test/neural/schema_grammar_narrated.ring

`constrained_decoding_narrated` 50 assertions
base/test/neural/constrained_decoding_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en outils. Les ateliers de langage restent en avance sur les axes auxquels la conception du domaine de projets a délibérément renoncé : l'outillage projectionnel (des éditeurs qui comprennent un langage pendant que vous tapez), l'intégration aux IDE, la profondeur macro-théorique. L'étude du domaine de projets le dit dans ses propres mots : c'est un pari, et il est nommé comme tel.

Le terrain a construit le décodage contraint le premier. L'échantillonneur de Softanza suit l'algorithme de l'implémentation de référence et n'embarque rien. Mesurées sur un seul petit modèle de cent trente-cinq millions de paramètres, sur dix consignes, les réponses étaient valides du premier coup deux fois sur dix quand on les vérifiait après coup. Tenu à la grammaire, le modèle était valide dix fois sur dix. C'est un seul petit modèle et dix consignes, et rien dans le dossier n'en mesure un grand.

Et l'âge coupe dans les deux sens. Le code généré rend aussi bon marché la fabrication négligente de langages. C'est pourquoi le tribunal n'est pas un accessoire d'un langage déclaré. Il en est le cœur. C'est mon point de vue.

7. LA LIMITE

CONSTRUIT — le vocabulaire de la couche naturelle comme données en cinq langues, avec ses refus ; le tribunal de la sortie structurée, son compilateur de grammaire et son échantillonneur ; les langages de motifs, qui acceptent un mot qu'ils ne connaissent pas

CONÇU — la forge qui produit l'analyseur, le tribunal et la grammaire d'un langage à partir de sa seule déclaration ; une grammaire émise pour chaque langage déclaré ; le style d'écriture comme grammaire déclarée

ABSENT — une couche naturelle fermée jusqu'au fond : une forme de liste atteint encore le code de l'hôte ; des classes de motifs qui refusent un mot qu'elles ne connaissent pas ; la lettre ɗ du hausa standard dans le paquet hausa

Ce qui est construit : la couche naturelle, son vocabulaire comme données en cinq langues, l'anglais, le hausa, le français, l'arabe et le turc, et ses refus. Construits aussi : le tribunal de la sortie structurée avec son compilateur de grammaire et son échantillonneur, et les langages de motifs. Ceux-ci acceptent aujourd'hui un mot qu'ils ne connaissent pas ; les fermer est à venir.

Ce qui est conçu : la forge qui produirait, à partir de la seule déclaration d'un langage, son analyseur, son tribunal, son imprimeur (la partie qui réécrit une phrase) et sa grammaire pour un modèle. La forge est construite hors de la bibliothèque et conçue pour la bibliothèque : une première esquisse en a été

construite hors de la bibliothèque en octobre 2026. La bibliothèque ne peut pas encore l'exécuter. Le texte de la vision dit que la forge n'a pas encore forgé (fabriqué les parties d'un langage à partir de sa déclaration). L'esquisse a depuis forgé des tribunaux et des grammaires pour plusieurs langages, jugés par les propres fixtures de ces langages, et la vision ne l'a pas rattrapée. Le chapitre 12 la raconte avec ses limites. Une grammaire pour chaque langage déclaré est conçue et à venir.

Ce qui est absent, et la loi l'énonce. La couche naturelle n'est pas fermée jusqu'au fond. Une forme, une liste écrite entre crochets, est encore remise à l'hôte pour être évaluée. Une fonction de l'hôte à l'intérieur des crochets s'exécute-t-elle ? Essayez-en une :

HARO

```
? @@( Naturally("Create a list with [ upper('x'), 1 ] and remove its duplicates").Result() )
#--> [ "X", 1 ]
```

✓ exécuté : a affiché ce que ce bloc promet

La fonction `upper` n'est pas un mot du dictionnaire naturel, et la liste est revenue `[ "X", 1 ]`. Elle s'est exécutée quand même ! Le dossier connaît déjà cette échappée et a acheminé sa réparation. Tant que la réparation n'est pas arrivée, « la couche naturelle est un langage fermé » est vrai de ses mots et pas de ses listes, et ce livre le dira.

Et le paquet hausa ne lit pas encore la lettre ɗ, qu'emploie l'orthographe standard du hausa. Le paquet la stocke dans un encodage qui ne la porte pas encore. Quand je l'ai sondé le 8 octobre 2026 sur l'instantané précédant ce commit, une phrase écrite avec cette lettre n'était pas lue, et le paquet ne le signalait pas. Le hausa de ce chapitre est écrit sans le crochet, comme le paquet le lit aujourd'hui. Le dossier ne tient aucune lecture des phrases du paquet par un locuteur du hausa. Les phrases en langue étrangère imprimées dans ce chapitre viennent de la narration et des gardes de la bibliothèque, ou sont composées à partir de son vocabulaire. Aucun locuteur natif ne les a lues pour autant que le dossier le montre.

Voilà les limites ; la section suivante ouvre six portes sur elles.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous continuez d'écrire du code. Ce que vous déclarez est une grammaire fermée, et le tribunal juge votre phrase et celle d'un agent d'IA par les mêmes règles.
L'analyste	Les mots de votre domaine deviennent un paquet de données, et le système les comprend ou les refuse avec la raison.
Le concepteur	Les refus font partie du langage : ce qui est hors liste est refusé en mode strict, donc il n'a jamais besoin d'un avertissement.
Le décideur	Un modèle d'IA tenu à une grammaire déclarée répond dans la bonne forme. Il peut encore répondre une fausseté dans la bonne forme, et le dossier imprime cette limite avec le résultat.
Le formateur	Une phrase en hausa, en français ou en arabe s'exécute et redit ce qu'elle a compris, de sorte qu'un apprenant lit la lecture que la machine fait de ses propres mots.
L'agent	Vous ne pouvez parler que le langage déclaré. Le mode strict est votre posture, et votre brouillon est, par la règle de la vision (conçue), une proposition que le tribunal juge.

9. EXERCICE

Déclarez, avec `StzOutputSchemaQ`, un enregistrement de trois champs tiré d'un domaine que vous connaissez : une demande de prêt, par exemple, avec un montant dans une fourchette, une devise tirée d'une liste fermée, et un objet en texte libre. Avant de rien exécuter, écrivez trois réponses qu'un modèle d'IA pourrait donner : une que le tribunal doit accepter, une qu'il doit refuser à cause de sa forme, et une qu'il acceptera bien qu'elle soit fausse. Sous chacune, écrivez votre prédiction comme une promesse.

Exécutez-le. La troisième réponse est la leçon de ce chapitre. Écrivez, en une phrase, qui dans votre domaine doit la juger, puisque la grammaire ne le peut pas.

Je vous laisse celui-ci. La section 7 dit qu'une liste écrite entre crochets est encore remise à l'hôte, et la section 4 dit pourquoi un langage doit refuser toute forme qui atteint le code de l'hôte. Exécutez le bloc de la section 7 avec un seul changement, une autre fonction de l'hôte à l'intérieur des crochets :

HARO

```
? @@( Naturally("Create a list with [ lower('X'), 1 ] and remove its duplicates").Result() )
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Regardez ce que la bibliothèque fait de la fonction, et dites en une phrase ce qu'aurait fait une couche naturelle fermée. Quand vous l'aurez fait, vous saurez pourquoi « la couche naturelle est un langage fermé » est vrai de ses mots et pas encore de ses listes.

Le chapitre 12 est *Un langage, déclaré*. Son principe : déclarer un langage est aussi facile que déclarer une variable.

OÙ SONT LES PREUVES

NARRATION	stznatural-vs-ring-naturallib.md
NARRATION	stz-near-natural-language-programming.md
NARRATION	stz-bridging-minds-and-code.md
NARRATION	stzlistex-pattern-matching-for-lists.md
SUR LE SITE	languages
DÉCISIONS DE L'ÉTUDE	12.4, 12.10, 12.16, 12.19, 12.21, 12.27, 12.28, 12.29, 12.30

PARTIE III · CHAPITRE 12

Un langage, déclaré

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 8 bloc(s) sur 8 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends une seule déclaration de la commande d'un restaurant et je la donne à trois lecteurs : une personne, un modèle d'IA et le tribunal. Je laisse la même déclaration produire une grammaire, et je montre où la grammaire et le tribunal s'accordent et où ils se séparent. Puis je quitte la bibliothèque pour la forge. C'est la machinerie qui transforme la seule déclaration d'un langage en son tribunal et sa grammaire, et je dis ce qu'elle a forgé et ce qu'elle n'a pas forgé. J'ai décidé que Haro est la langue des langues, et j'ai refusé une forme d'échappatoire ouverte. Le principe est une règle de Haro : déclarer un langage est aussi facile que déclarer une variable.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Deux familles d'outils permettent de déclarer un langage sur le terrain. La première est l'atelier de langages et le générateur d'analyseurs : JetBrains MPS, Xtext, Spoofax, et le `#lang` de Racket vous donnent de quoi définir un langage avec sa grammaire, son éditeur et ses vérifications. L'étude de terrain du domaine de projets lui-même, d'août 2026, dit pourquoi ils n'ont pas gagné. Ils dépendaient d'un écosystème, et les langages de Racket vivent et meurent à l'intérieur de Racket.

La seconde est le DSL interne : une bibliothèque dont les appels se lisent comme des phrases, alors que le fichier peut encore contenir n'importe quel code hôte. La même étude a la phrase que je retiens : la fermeture est le produit, et un DSL interne qui ne

peut pas refuser du code hôte arbitraire est une API qui porte le costume d'un langage. Son groupe témoin, ce sont les langages fermés qui ont gagné : SQL, regex, Make, HTML et CSS, HCL, GraphQL, Markdown, CUE.

Pour la grammaire qu'un modèle doit respecter, le dossier nomme un seul outil : l'échantillonneur de grammaire d'une bibliothèque d'inférence locale bien connue, que la machine de la bibliothèque suit pour son encodage des éléments et son algorithme. Rien n'en est repris. C'est une connaissance générale, pas une étude du dossier, que les fournisseurs de modèles proposent des modes de sortie structurée qui prennent un schéma JSON, et que des bibliothèques ouvertes transforment un schéma ou une expression régulière en masque sur les jetons du modèle.

Le chapitre 11 a dit ce qu'est un langage fermé, et ce qui a été perdu quand le terrain l'a laissé ouvert. Je le tiens pour acquis. Ce chapitre dit ce qui est nouveau : ce qui *produit* un langage fermé.

2. LE PREMIER PRINCIPE

Déclarer un langage est aussi facile que déclarer une variable.

C'est l'intitulé d'une règle de Haro, tranchée le 1er octobre 2026, et le noyau de la vision dit la même chose : l'époque rend la déclaration d'un langage aussi facile que celle d'une variable. L'autre moitié du chapitre a sa phrase, tirée de la doctrine de totalité de l'alphabet, ratifiée le 27 septembre : la malformation meurt à l'échantillonneur, la fausseté meurt au tribunal, et la non-terminaison est inexprimable.

Et celle que je dois porter à côté de chaque gain, tirée du contrat de la bibliothèque pour la sortie structurée : la structure tue la malformation, et non la fausseté.

3. LA FORME SOFTANZA

Prenons une déclaration de ce à quoi doit ressembler la commande d'un restaurant. Elle a trois champs : un plat pris dans un menu fermé, une quantité entre 1 et 12, et une note facultative. Un seul texte est lu par trois lecteurs. Une personne lit une

description. Un modèle lit une clause pour son prompt. Et le tribunal lit le même texte et juge une réponse. Que répond-il à un plat nommé pizza et à quarante portions ? Exécutez-le :

HARO

```
oO = StzOutputSchemaQ([
        [ :field = "dish",     :type = :oneof,  :choices = [ "margherita", "tiramisu" ] ],
        [ :field = "quantity", :type = :number, :must = [ [ ">=", 1 ], [ "<=", 12 ] ] ],
        [ :field = "note",     :type = :string, :optional = 1 ]
])
oO {
        ? Describe()
        #--> schema 'output' -- 3 field(s)
        #      dish : oneof (margherita | tiramisu) [required]
        #      quantity : number [required], must be at least '1', must be at most '12'
        #      note : string [optional]
        ? PromptClause()
        #--> Answer with ONLY this structure, one field per line, nothing before it and nothing
after it:
        #
        #    dish: <one of: margherita | tiramisu>
        #    quantity: <number>
        #    note: <text>
        aV = Verify([ :dish = "pizza", :quantity = 40 ])
        ? aV[:ok]
        #--> 0
        ? aV[:findings][1][:message]
        #--> field 'dish' is a closed enumeration (margherita | tiramisu) and the model answered
'pizza'.
        ? aV[:findings][2][:message]
        #--> field 'quantity' must be at most '12', and the model answered '40'.
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le tribunal répond dans les mots de la déclaration : `field 'dish' is a closed enumeration (margherita | tiramisu) and the model answered 'pizza'.` Maintenant la même déclaration produit une grammaire. Le tribunal et la grammaire naissent d'un seul texte, et ils s'accordent sur la forme. Ils se séparent sur une valeur. Quarante portions ont la bonne forme : lequel des deux les laisse passer, et la bibliothèque dit-elle ce que la grammaire a laissé tomber ? Exécutez-le :

HARO

```
cG = oO.ToGBNF()
cGood  = "dish: margherita" + nl + "quantity: 2" + nl + "note: no basil" + nl
cBand  = "dish: margherita" + nl + "quantity: 40" + nl + "note: no basil" + nl
cShape = "dish: margherita" + nl + "quantity: many" + nl + "note: no basil" + nl
? StzGrammarAccepts(cG, cGood)
#--> 1
? oO.Accepts(cGood)
#--> 1
? StzGrammarAccepts(cG, cBand)
#--> 1
? oO.Accepts(cBand)
#--> 0
? StzGrammarAccepts(cG, cShape)
#--> 0
? oO.Accepts(cShape)
#--> 0
? len(StzFind("lessequal", oO.UnenforcedByGrammar())) > 0
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

La deuxième paire est l'enjeu du chapitre : la grammaire répond `1` et le tribunal répond `0` ! Je n'imprime pas la liste de ce que la grammaire a laissé tomber : son texte porte, à un endroit, l'ancien nom du tribunal, et ce nom appartient à l'histoire de l'annexe H. Je vérifie que la liste nomme la borne de l'intervalle.

On ne demande pas au modèle d'IA d'être prudent. L'échantillonneur demande à la grammaire, à chaque jeton candidat, si elle pourrait continuer. Ainsi un mot qui n'est pas au menu n'est pas refusé après avoir été dit. Peut-il seulement commencer ?

HARO

```
? StzGrammarCouldContinue(cG, "dish: t")
#--> 1
? StzGrammarCouldContinue(cG, "dish: p")
#--> 0
? StzGrammarCouldContinue(cG, "dish: margherita" + nl + "quantity: 2")
#--> 1
? StzGrammarCouldContinue(cG, "dish: margherita" + nl + "quantity: <")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Il ne le peut pas : `dish: p` répond `0`. La dernière ligne est un échec que la mesure de la section 5 a trouvé dans un petit modèle : le substitut `<number>` répété là où doit se trouver un nombre. Et un jeton n'est pas un octet. Un morceau peut être valide comme préfixe et invalide comme complétion, et la machine reçoit le morceau entier. Que répond-elle à `yesterday`, qui commence comme `yes` ?

HARO

```
cY = 'root ::= "yes" "\n"'
? StzGrammarCouldContinue(cY, "y")
#--> 1
? StzGrammarCouldContinue(cY, "ye")
#--> 1
? StzGrammarCouldContinue(cY, "yesterday")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

`y` et `ye` répondent `1`, et `yesterday` répond `0` !

Ce que la machine ne tiendra pas, elle le refuse en le nommant. Un langage fermé refuse ce qu'il ne peut pas porter, et n'approxime jamais. Une règle récursive à gauche est refusée par un code, de même qu'une règle citée et jamais définie. Une structure imbriquée est refusée par la grammaire. Le tribunal la lit-il quand même ?

HARO

```
? left(StzGrammarRefusal('root ::= tail' + nl + 'tail ::= tail "x" | "x"'), 7)
#--> GBNF-M7
? left(StzGrammarRefusal('root ::= greeting'), 7)
#--> GBNF-M4
? StzGrammarRefusal(cG) = ""
#--> 1
StzOutputSchemaQ([
        [ :field = "title",  :type = :string ],
        [ :field = "author", :type = :structure, :fields = [ [ :field = "name", :type = :string ]
] ]
]) {
        ? IsExpressibleAsGrammar()
        #--> 0
        try
                ToGBNF()
        catch
                ? len(StzFind("GBNF-F5", cCatchError)) > 0
        done
```

```
        #--> 1
        ? ParseOutput('{ "title":"T", "author":{"name":"Ada"} }')[:ok]
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le texte du refus de la structure imbriquée porte lui aussi l'ancien nom du tribunal, donc je n'imprime que les codes. La dernière ligne est l'enjeu : le tribunal lit ce que la grammaire refuse d'écrire.

Le langage de déclaration est lui-même fermé. Une faute de frappe dans la loi est refusée au moment où la loi est déclarée. Le contrat de la bibliothèque dit pourquoi. Une clé comme `:requird = 0` laisserait un champ obligatoire sans un mot, et c'est la classe de faute de frappe qu'un tribunal ne doit pas tolérer dans sa propre loi. Que répond le tribunal à une telle clé ?

HARO

```
try
        StzOutputSchemaQ([ [ :field = "x", :type = :string, :requird = 0 ] ])
catch
        ? cCatchError
done
#--> stzOutputSchema: unknown key ':requird' in field #1 (use :field, :name, :type, :required,
:optional, :choices, :oneof, :of, :fields, :must, :note, :description).
```

✓ exécuté : a affiché ce que ce bloc promet

La faute de frappe a été refusée sur-le-champ, et le message liste les clés permises !

Et une fonction hôte n'est pas un mot du langage fermé des conditions. Il y a deux façons de dire une condition, le langage fermé et la porte d'une fonction. Une racine carrée est un mot du langage d'expressions du moteur, et un sinus n'en est pas un. Que répond la forme textuelle au sinus, et que répond la forme fonction, qui exécute du code hôte ?

HARO

```
StzListQ([ 4, 7, 10, 3, 8 ]) {
        ? @@( FindW('{ sqrt(@item) < 3 }') )
```

```
        #--> [ 1, 2, 4, 5 ]
        ? @@( FindW('{ sin(@item) < 2 }') )
        #--> [ ]
        ? @@( FindWF( func x { return sin(x) < 2 } ) )
        #--> [ 1, 2, 3, 4, 5 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Ce que j'ai vu, et que la narration ne dit pas : le refus est silencieux et entier. La forme textuelle a imprimé `[ ]` et la forme fonction a imprimé `[ 1, 2, 3, 4, 5 ]` : rien de l'une, tout de l'autre. Une condition dont la branche gauche est vraie et dont la droite porte un sinus a imprimé `[ ]`. Une condition qui appelle une fonction de moi ne l'a jamais appelée. Une erreur de syntaxe a aussi imprimé `[ ]`. Le langage est fermé, et il refuse sans un mot. La couche naturelle du chapitre 11 dit pourquoi elle refuse, et celle-ci ne le dit pas.

LA FORGE, HORS DE LA BIBLIOTHÈQUE

Un langage déclaré une fois devrait donner son tribunal, sa bouche, sa voix et ses lecteurs. C'est la forge, et c'est là que l'étude de ce livre rencontre sa propre limite : la forge n'est pas dans la bibliothèque.

Je dirai ce que le dossier contient, et ce que j'ai observé le 8 octobre 2026 sur le runtime plus ancien, l'oracle. Le dossier est une charte et un spike dans un autre dépôt, en pur code du noyau qui tourne sur l'oracle. La forge lit une déclaration et écrit deux fichiers, un programme de tribunal et une grammaire. Les échelons du spike, entre le 1er et le 4 octobre 2026, ont forgé des tribunaux et des grammaires pour le langage machine du domaine de projets et pour un langage de flotte déclaré par-dessus. Ils ont fait de même pour le langage d'expressions et pour quatre lettres de plus de l'alphabet, la famille de langages déclarés de la vision, chacun connu par une lettre. Chacun a été jugé sur les cas propres du langage préexistant. Ces cas étaient sans adaptation pour le langage d'expressions (W) et pour Z, réexprimés pour F et A, et, pour Q, les réponses propres de l'ancien langage, relevées. Dans l'ordre : 117 sur 120 pour le langage machine, les trois restants étant les sortes du

fichier de flotte. Puis 32 sur 32 pour le langage de flotte, 29 sur 29 pour W, 16 sur 16 pour Z, 25 sur 25 pour Q, 26 sur 26 pour F et 35 sur 35 pour A. C'est la méthode du chapitre 3, déclarer puis gouverner, appliquée à la forge elle-même.

J'ai fait tourner la forge sur un petit langage que j'ai écrit pour ce livre, la commande d'une table dans les mots de la salle à manger. C'est un exemple, écrit pour le livre et exécuté une fois, hors de la bibliothèque :

```
DEFINE LANGUAGE order AS ( VERSION "0.2", GRAMMAR "0.2", COURT "none", EXTENSION ".order" )
  RATIONALE "A table's order, in the words of the dining room"
DEFINE MENU dishes AS ( WORDS [margherita, tiramisu, chianti],
  MESSAGE "the kitchen cooks margherita, tiramisu or chianti" ) RATIONALE "The kitchen's
closed menu"
DEFINE RANGE portions AS ( MIN 1, MAX 12,
  MESSAGE "{value} portions is not an order: one to twelve" ) RATIONALE "A table orders
between one and twelve"
DEFINE DECLARATION line AS ( VERB "ORDER", SEATS [dish: dishes, quantity: portions, note:
text?] )
  RATIONALE "One line of an order: what, how many, and a word for the kitchen"
DEFINE REFUSAL unknown_verb AS ( MESSAGE "The dining room speaks only of orders." )
  RATIONALE "The closure speaks when an alien verb arrives"
```

La forge a imprimé qu'elle avait forgé une sorte et un refus, en une fraction de seconde d'horloge, et a écrit un tribunal d'environ 2,6 kilooctets. Que répond ce tribunal à un plat hors du menu, à quarante portions et à un verbe étranger ? Il a refusé les trois, dans les mots propres de la déclaration :

```
DISH pizza          ->  order (line 2): the kitchen cooks margherita, tiramisu or chianti,
not 'pizza'
QUANTITY 40         ->  order (line 3): 40 portions is not an order: one to twelve
DROP TABLE orders   ->  order (line 1): Unknown verb 'DROP': The dining room speaks only of
orders.
```

Les refus sont les mots propres de la déclaration, écrits à côté de la grammaire. Même `DROP TABLE orders` a été refusé dans les mots de la salle à manger ! C'est toute la preuve du langage `order` : il a été exécuté une fois, hors de la bibliothèque,

et ce n'est pas une déclaration du dossier.

Trois observations de plus, chacune une limite.

Le méta-tribunal refuse l'échappatoire. La déclaration du langage des déclarations liste les opérateurs et les opérandes qu'elle peut employer. Une déclaration qui listait `eval` parmi ses opérateurs, dans une fixture épinglée le 29 août 2026, a été refusée : `eval` n'est pas dans le menu fermé des opérateurs, une échappatoire est inexprimable, non interdite. Un appel hôte parmi les opérandes a été refusé de la même façon.

La grammaire de la forge ne tient que la forme. Un siège de menu ressort comme un identifiant général, donc la grammaire forgée laisse passer `pizza`. Le compilateur de schémas de la bibliothèque, dans les blocs ci-dessus, ferme le même menu en une alternance. Sur le langage machine du domaine de projets, la grammaire forgée accepte une référence pendante que le tribunal refuse. Le dossier nomme une différence entre la bouche et le tribunal, la part qu'une grammaire hors contexte ne peut pas dire : les noms en double, les références pendantes. Je n'ai trouvé aucun enregistrement qui énonce le manque du menu comme un manque.

La vision n'a pas rattrapé. Le texte de la vision, daté entre le 27 septembre et le 5 octobre 2026, dit que la forge n'a pas forgé. Le fichier des principes, daté du 1er octobre 2026, dit que la forge n'existe pas. Le même dépôt a construit le spike le même jour et forgé pour plusieurs langages. La phrase vraie a donc deux côtés. La forge a forgé des tribunaux et des grammaires pour plusieurs langages, hors de la bibliothèque, et elle n'a pas forgé la liste complète que le principe promet. Et le texte de la vision n'a pas été mis à jour. Dans les mots d'étape du livre, la forge est construite hors de la bibliothèque et conçue pour la bibliothèque, et la limite du chapitre 11 le dit de même.

LES PARTIES D'UN LANGAGE DÉCLARÉ

La charte liste ce que la forge produit à partir d'une déclaration, ses parties, et je donne à chacune un mot d'étape :

PARTIE	CE QUE C'EST	ÉTAPE
le texte	la déclaration, simple et fermée, avec une justification exigée sur chaque déclaration	construit à l'extérieur ; l'analogue de la bibliothèque est la déclaration de schéma
le tribunal	le juge généré pour le langage, qui parle dans les mots propres de la déclaration	construit à l'extérieur ; l'analogue de la bibliothèque est `Verify` et `Accepts`
la bouche	la grammaire pour un modèle d'IA contraint	construit dans la bibliothèque pour les schémas ; construit à l'extérieur pour les langages déclarés, la forme seulement
le prompt	l'instruction qu'un modèle d'IA lit, réglée par la mesure	construit dans la bibliothèque ; inconnu pour les langages déclarés
la lecture de la personne	une description pour un humain	construit dans la bibliothèque (`Describe`)
la voix	le langage qui narre une exécution, une phrase, un refus ; jamais stocké et jamais relu par l'analyse	construit à l'extérieur ; l'analogue de la bibliothèque est `Understood()`
les langues	une langue humaine comme fichier de données	construit à l'extérieur pour deux langues ; construit dans la bibliothèque pour cinq paquets (chapitre 11)
les projections	le monde rendu à une application, un serveur, une page, un appareil	conçu
les facettes	une seconde syntaxe de Haro	conçu ; le chapitre 16 la raconte

Le mot *facette* a deux sens dans ce livre, et je définis ici celui de cette partie. Une facette est une seconde syntaxe de Haro, pour un programmeur d'un autre langage : une autre graphie de la grammaire unique, et jamais un runtime. C'est la dernière ligne du tableau, et le chapitre 16 la raconte. L'autre sens est un onglet du ruban de l'environnement, une phase d'une vie, celui du chapitre 26.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé que Haro est la langue des langues. L'en-tête de la charte me cite, du 1er octobre 2026 : Haro est une langue des langues, dans laquelle les artisans du logiciel définissent un langage pour leur domaine et programment avec lui, ou laissent leurs agents d'IA le faire. La charte enregistre un renversement : sa première décision, en tant que socle généraliste sous la famille des langages déclarés, a été renversée. Le reste de cette ratification était mon « suivez vos recommandations » sur sept décisions, et ce sont des décisions par délégation.

J'ai décidé de déclarer, et non d'écrire. L'acte du noyau : déclarer un langage. Ce fut une extraction et non une invention : les premiers langages ont été déclarés après qu'ils tournaient déjà et avaient des consommateurs.

J'ai refusé une forme d'échappatoire ouverte. Aucune clause, aucun opérateur ni opérande ne désigne du code hôte, des appels, une évaluation ou des entrées et sorties. Inexprimable, non interdit. Deux fixtures épinglent le refus, et je les ai relancées.

J'ai refusé la logique dans la surface de déclaration. La déclaration déclare et ne calcule pas. L'alphabet n'est délibérément pas Turing-complet, et c'est sa puissance et non sa pauvreté. C'est mon point de vue.

J'ai refusé le langage naturel comme surface de déclaration. Le langage naturel demande, et il ne déclare jamais. Un brouillon peut être proposé, et le tribunal décide, et l'humain ratifie.

J'ai refusé un seul texte pour déclarer et pour calculer. La programmabilité et la fermeture n'occupent jamais la même surface. Le code de calcul n'atteint un monde déclaré que par les portes que la forge génère, et le chapitre 16 parle de la porte.

J'ai refusé de laisser la moitié grammaire mettre le tribunal à la retraite. Le contrat dit que le tribunal ne part pas à la retraite quand le décodage contraint arrive. Et ce que la grammaire ne peut pas exprimer, elle le refuse, et ne l'aplatit ni ne l'approxime jamais.

J'ai refusé une seconde facette comme un second sens. Une facette épelle le graphe ; elle ne lui donne jamais le sens de Python ou de JavaScript. C'est le mémo d'une session, qui propose et ne tranche rien, et sa question de savoir combien de facettes avant l'annonce est la mienne, et sans réponse dans le dossier.

Vient ensuite le tribunal, et les gardes qui sont derrière lui.

5. LE TRIBUNAL

`structuredoutput_narrated` prouve qu'une déclaration est jugée quand elle est déclarée, avec un type inconnu, une clé inconnue et un menu vide refusés. Elle prouve que deux formes de transmission sont lues et que rien n'est deviné, qu'il n'y a pas de crédit partiel, que l'imbrication fonctionne, et qu'une nouvelle tentative tient dans un seul budget. Dans sa dixième scène, elle prouve qu'un mensonge est laissé passer exprès. `schema_grammar_narrated` prouve que la déclaration se compile en grammaire, et qu'une énumération fermée devient une alternance fermée. Elle prouve que l'optionnel et l'arité des listes sont traités, que les contraintes non imposées sont listées par champ et par opérateur, et qu'une structure imbriquée est refusée avec un code nommé. `constrained_decoding_narrated` prouve que la machine parcourt une grammaire octet par octet, qu'un préfixe n'est pas un tout, et que `yesterday` est refusé. Elle prouve que le substitut écrit par le modèle ne peut pas être atteint, que les refus sont nommés, que la structure n'est pas la vérité, et qu'une grammaire sur une session en flux est refusée. Ses deux scènes qui demandent un modèle génératif le disent.
`483_w_and_wf_forms_in_conditional_code` prouve que les deux formes d'une condition donnent les mêmes positions. Elle n'affirme pas le refus du langage fermé, et aucune garde ne l'affirme.

Aucune garde de la bibliothèque ne teste une forge, puisqu'il n'y en a pas. La garde de la forge est à l'extérieur : le spike passe vingt-trois étapes en 367,5 secondes le 4 octobre 2026, d'après son propre fichier. Je ne l'ai pas lancé. Et la première exécution du méta-tribunal lui-même fut de vingt-huit sur vingt-huit le 29 août 2026. Le lecteur de gardes du livre ne résout que les gardes de l'arbre de tests de la bibliothèque, donc celles-ci ne peuvent pas être nommées dans le front matter.

Le premier essai est valide dans deux cas sur dix quand on vérifie après coup, et dans dix sur dix quand il est contraint. Ce chiffre est un nombre dans un commentaire de ces gardes. On ne peut pas le reprendre ici : les scènes réelles demandent un fichier de modèle génératif, et l'instantané ne contient, dans son dossier des modèles, qu'un fichier de somme de contrôle. Le chapitre 11 porte le chiffre, et j'y renvoie.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`structuredoutput_narrated` 73 assertions
base/test/neural/structuredoutput_narrated.ring

`schema_grammar_narrated` 28 assertions
base/test/neural/schema_grammar_narrated.ring

`constrained_decoding_narrated` 50 assertions
base/test/neural/constrained_decoding_narrated.ring

`483_w_and_wf_forms_in_conditional_code` 2 assertions
base/test/list/483_w_and_wf_forms_in_conditional_code.ring

`calc_column_engine_narrated` 14 assertions
base/test/table/calc_column_engine_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les ateliers de langages mènent sur l'outillage. L'étude de terrain le dit elle-même : ils restent supérieurs sur les axes que la conception abandonne délibérément, l'outillage projectionnel, l'intégration à l'éditeur et la profondeur des macros. Et le pari que ces axes sont les mauvais est nommé comme un pari.

Le code généré rend bon marché aussi la fabrication indisciplinée de langages, et c'est pourquoi le tribunal est le cœur et non un accessoire.

Le décodage contraint : le terrain l'a construit le premier, et l'implémentation de référence est celle d'une autre bibliothèque. Ce que le dossier ajoute, c'est l'appariement : une grammaire et un tribunal issus d'une seule déclaration, et la liste de ce que la grammaire a laissé tomber.

Une grammaire sur-approxime un tribunal. Les noms en double, les références pendantes et la cohérence d'un menu ne sont pas hors contexte. Je l'ai observé dans les deux sens plus haut : une plage que la grammaire ne peut pas porter, et une référence qu'elle ne peut pas porter.

Le code qu'une forge émet doit être lu par quelqu'un. Le style de famille dont hérite chaque langage déclaré a un test de réussite : apprendre le deuxième langage devrait prendre une soirée, parce que c'est le premier langage avec d'autres verbes. C'est une exigence de conception et non une mesure. Aucun apprenant n'a été chronométré dans le dossier que j'ai lu.

La limite vient ensuite, au bord d'un second langage fermé.

7. LA LIMITE

CONSTRUIT

dans la bibliothèque : une déclaration jugée au moment où elle est déclarée ; un tribunal et une grammaire issus d'une seule déclaration, avec la liste de ce que la grammaire a laissé tomber ; une machine à grammaire qui parcourt les octets et refuse en nommant ; une description pour une personne et un prompt pour un modèle d'IA à partir du même texte. Hors de la bibliothèque : une forge qui transforme une déclaration en tribunal et en grammaire, exécutée sur les propres langages du domaine de projets

CONÇU	la forge à l'intérieur de la bibliothèque ; sa liste complète de produits issus d'une seule déclaration (imprimeur, graphe de connaissance, porte naturelle, narration, runtime, un seul binaire statique) ; les projections d'un langage vers une application
ABSENT	la grammaire de la forge pour un mot de menu ; une garde qui affirme le refus d'une fonction hôte dans une condition ; le chiffre deux sur dix contre dix sur dix, qui demande un fichier de modèle que l'instantané ne contient pas

La bibliothèque a deux langages de formules fermés, et ils se comportent différemment au bord. Un lecteur qui croit que le langage fermé des conditions, celui du chapitre 11 et de la condition ci-dessus, couvre toutes les formules se trompera à table. La colonne calculée d'un tableau retombe, d'après son propre en-tête, sur le chemin d'évaluation classique quand le moteur ne peut pas compiler une formule. Le mot `sin` que la condition a refusé s'exécute-t-il ici ?

HARO

```
o1 = new stzTableAggregator([
        [ :A, [ 0, 1, 4 ] ],
        [ :B, [ 2, 4, 20 ] ]
])
o1.AddCalculatedCol(:SINE, 'sin(@(:A))')
? @@( o1.Col(:SINE) )
#--> [ 0, 0.84, -0.76 ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le sinus que la condition a refusé, le tableau l'a calculé, et il a imprimé `[ 0, 0.84, -0.76 ]` ! J'ai aussi vu, et je ne le montre pas, une seconde chose. Sur une autre table d'essai, avec A égal à 1, 3 et 10 et B égal à 2, 4 et 20, j'ai écrit une formule `@(:A) + HostBonus(@(:B))`. La fonction de moi qui s'y trouve multiplie par mille. La formule a imprimé `[ 2001, 4003, 20010 ]` : une fonction hôte définie par l'utilisateur s'est exécutée de l'intérieur d'une chaîne de formule. Et une formule qui nomme une fonction qui n'existe pas a levé l'erreur propre de l'hôte. Le second langage fermé est donc ouvert à son bord. Aucune garde n'affirme l'un ou l'autre comportement.

Ce qui est construit : dans la bibliothèque, une déclaration jugée quand elle est déclarée ; un tribunal et une grammaire issus d'elle ; la liste de ce que la grammaire a laissé tomber. Une machine à grammaire qui dit qu'elle est contrainte ; un prompt et une description issus du même texte. Le build compte 151 assertions dans trois gardes narrées. Deux petits langages fermés de conditions, avec des limites. Les langues de la couche naturelle comme données.

Ce qui est construit, mais hors de la bibliothèque : une forge d'une déclaration vers un tribunal et une grammaire, dans le dépôt de la charte de Haro.

Ce qui est conçu : la forge qui produit, à partir d'une seule déclaration, l'analyseur, le tribunal, l'imprimeur, la grammaire, le graphe de connaissance, la grammaire de requête, la porte naturelle, la narration, le runtime et un binaire statique. D'après ma lecture de la charte, le spike a montré le tribunal, la grammaire, un runtime qui parcourt la déclaration, une grammaire de requête, une narration et le tribunal de la lettre naturelle. Je n'ai pas trouvé montré un imprimeur pour les phrases déclarées, un graphe de connaissance forgé à partir d'une déclaration, ni le binaire statique unique par langage. La bibliothèque qui a une forge : rien en elle ne déclare ni ne forge un langage. Une grammaire pour chaque langage déclaré livrée dans la bibliothèque : promise dans le noyau, non livrée. Et les projections : le monde rendu à une application, un serveur, une page, un appareil.

Ce qui est absent : la grammaire de la forge pour un mot de menu, que j'ai vue ne tenir que la forme, plus haut, et le chiffre de deux sur dix contre dix sur dix, repris. Et tout enregistrement que la forge tourne sur le runtime de Haro. Au moment où la vision a été écrite, le langage n'existait pas ; les exécutions de la charte sont sur le runtime plus ancien et sur l'oracle.

Les six portes suivent, du programmeur à l'agent.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous déclarez la forme en quelques lignes et vous en tirez son tribunal, sa grammaire et son prompt. Ce que vous n'obtenez pas, c'est un moyen d'exécuter une fonction hôte depuis la déclaration : le langage n'a pas de mot pour cela. Le langage de formules du tableau, à la section 7, est un autre langage, et ouvert à son bord.
L'analyste	La déclaration est du texte brut dans les mots du domaine, et le tribunal répond dans ces mots.
Le concepteur	Le refus est déclaré à côté de la grammaire, dans la voix du domaine, et n'est pas rédigé par runtime. Un concepteur choisit ce que dit un menu fermé quand il est rompu.
Le décideur	Une seule déclaration produit la bouche et le tribunal, et la forge elle-même publie où ils sont en désaccord. La forge est aujourd'hui hors de la bibliothèque. Ne l'achetez pas comme livrée.
Le formateur	Le lecteur peut prédire, octet par octet, ce que l'échantillonneur permettra, et apprend qu'une grammaire tue la malformation et non la fausseté.
L'agent	Vous ne parlez que la grammaire qu'on vous a remise. Une phrase que vous rédigez est une proposition que le tribunal juge. Votre bouche est la forme, et vos faussetés sont celles du tribunal et d'une personne.

9. EXERCICE

Écrivez, à la main, une grammaire pour une ligne de commande de déjeuner : `dish:` puis l'un de deux plats de votre choix, puis un saut de ligne. Avant de rien exécuter, écrivez vos prédictions comme des promesses :

HARO

```
cG = 'root ::= ...'     # your grammar here
? StzGrammarAccepts(cG, "dish: <your first dish>" + nl)
#--> ?
? StzGrammarAccepts(cG, "dish: <a dish not on your menu>" + nl)
#--> ?
? StzGrammarCouldContinue(cG, "dish: <the first letters of your first dish>")
#--> ?
? StzGrammarAccepts(cG, "dish: <the first letters of your first dish>")
#--> ?
```

```
? StzGrammarRefusal(cG) = ""
#--> ?
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Ajoutez ensuite un nombre à la ligne, `quantity:` et des chiffres, et écrivez une sixième prédiction : votre grammaire accepte-t-elle `quantity: 4000` ? Exécutez-la. La leçon est la troisième, la quatrième et la sixième. La machine sait déjà qu'un préfixe pourrait continuer et n'est pas encore une phrase, et elle ne peut pas savoir que le nombre est trop grand. Notez, en une phrase, qui dans votre restaurant doit refuser quarante portions, puisque la grammaire ne le peut pas.

Je vous laisse celui-ci. La section 3 termine sa série de conditions sur un refus sans un mot, et la section 4 dit pourquoi le calcul reste hors de la surface qui déclare. Prenez les deux lignes qui posent à la liste la même question sous les deux formes, et changez une seule chose : écrivez un autre nombre à la place du 2, le même dans les deux lignes. Je n'imprime rien dessous.

HARO

```
o = new stzList([ 4, 7, 10, 3, 8 ])
? @@( o.FindW('{ sin(@item) < 2 }') )
? @@( o.FindWF( func x { return sin(x) < 2 } ) )
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-les pour plusieurs nombres. Cherchez un nombre pour lequel les deux formes impriment la même chose, et dites si l'impression seule vous apprend laquelle des deux a refusé et laquelle a répondu. Quand vous l'aurez fait, vous saurez ce que coûte un langage fermé qui refuse sans un mot.

Le chapitre suivant passe à la phrase naturelle, dans deux directions et sous une seule discipline. Une phrase s'exécute parce que chacun de ses mots est dans un dictionnaire que la bibliothèque possède, et ce que la bibliothèque ne connaît pas, elle le refuse, avec la raison.

OÙ SONT LES PREUVES

NARRATION	`stz-conditional-code-narration.md`
NARRATION	`stznatural-vs-ring-naturallib.md`
NARRATION	`stznatural-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
SUR LE SITE	languages
DÉCISIONS DE L'ÉTUDE	12.10, 12.21, 12.27, 12.29, 12.30

PARTIE III · CHAPITRE 13

Naturel, et exécutable : deux directions, une seule discipline

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 17 bloc(s) sur 17 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je fais tourner les deux directions du langage naturel dans la bibliothèque : la chaîne, où vous êtes l'analyseur, et la phrase, où c'est la bibliothèque. J'imprime les refus des deux côtés, cinq paquets de langue comme données et un sixième ajouté à l'exécution, et le compte de ce que chaque paquet sait dire. Je dis ce que j'ai refusé, les marqueurs de référence et la chaîne non grammaticale entre autres, et où le tribunal propre du dictionnaire échoue à ce commit. Le principe tient en une phrase. Une phrase s'exécute parce que chacun de ses mots est dans un dictionnaire que la bibliothèque possède, et ce que la bibliothèque ne connaît pas, elle le refuse, avec la raison.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Dans le langage courant du terrain, « programmer en langage naturel » veut dire une seule chose : demander du code à un modèle d'IA, ou lui demander de faire la tâche, dans un prompt. Les outils sont des assistants de discussion et des compléteurs de code, d'un usage large depuis 2021 environ. Ce sur quoi la phrase s'exécute, c'est la conjecture du modèle. La même phrase peut donner un code différent d'un jour à l'autre, le modèle a besoin d'un réseau ou d'un runtime local, et une personne lit le résultat pour savoir ce qui a été compris. Le chapitre 11 l'a dit, et j'y renvoie.

Des pratiques plus anciennes existent, et le dossier les nomme. Inform 7, un langage dont les programmes sont des phrases anglaises, compilées d'un bloc, avec un modèle du monde et un moyen de dire *il* d'une phrase à l'autre. AppleScript, proche de l'anglais et difficile à écrire, parce que sa naturalité était en surface et non dans un modèle sémantique. Les langages naturels contrôlés, dont le dispositif le plus fort, dit le commentaire de la bibliothèque elle-même, est de montrer ce qui peut être dit ensuite. Une lecture de session du 30 août 2026 est qu'Inform 7 est un compilateur de texte entier avec contexte référentiel, et non un répartiteur de commandes. Les dates sont une connaissance générale, pas une étude du dossier.

Aucun d'eux n'est ce que le dossier revendique. La revendication est le contraire du prompt : une phrase s'exécute parce que chacun de ses mots est dans un lexique fermé, défini comme données, associé à un verbe que la bibliothèque documente. Tout le reste est refusé avec la raison. Et elle a deux directions, que la revue de conception de la bibliothèque met en tableau. Dans la première, la machine est l'analyseur. Dans la seconde, c'est l'humain.

2. LE PREMIER PRINCIPE

Une phrase s'exécute parce que chacun de ses mots est dans un dictionnaire que la bibliothèque possède. Ce que la bibliothèque ne connaît pas, elle le refuse, avec la raison.

Mon étude du dossier, et le chapeau de la page du site du 8 octobre 2026, le disent. Pour la seconde direction, la thèse de la revue de conception : dans le langage quasi naturel, l'humain est l'analyseur. Et la devise plus ancienne, que le chapitre 5 porte : ce que vous pensez est ce que vous écrivez.

3. LA FORME SOFTANZA

Je commence par la seconde direction, parce qu'elle est la plus ancienne.

L'HUMAIN EST L'ANALYSEUR

Une chaîne quasi naturelle est une phrase dont chaque maillon est un vrai appel. La phrase est « le mot haro est un mot latin en minuscules, d'une longueur de 4 lettres et avec seulement 2 voyelles ». Quand elle est fausse, que dit la chaîne ? Exécutez-la :

HARO

```
? TheWordQ("haro").IsAQ([ :Lowercase, :Latin, :Word
]).WithQ().ALengthQ().OfQ(4).UnitQ(:Letters).AndQ().OnlyQ(2).VowelNB()
#--> TRUE
? TheWordQ("haro").IsAQ([ :Lowercase, :Latin, :Word
]).WithQ().ALengthQ().OfQ(4).UnitQ(:Letters).AndQ().OnlyQ(1).VowelNB()
#--> FALSE
? Why()
#--> no: expected exactly 1, found 2
```

✓ exécuté : a affiché ce que ce bloc promet

La chaîne répond `no: expected exactly 1, found 2`. Les mots de degré sont des dispositifs, et la raison est conservée. Le bloc la lit avec le `Why()` de la console ; la chaîne garde aussi le sien, `WhyCheckFailed()`, que je n'appelle pas ici :

HARO

```
? Q("AnnIE").AtLeast(2).VowelNB()
#--> TRUE
? Q("AnnIE").AtMost(2).VowelNB()
#--> FALSE
? Why()
#--> no: expected atmost 2, found 3
```

✓ exécuté : a affiché ce que ce bloc promet

La chaîne refuse ce que son sujet ne peut pas faire. Elle parle le même lexique que la direction de la machine : un mot français fonctionne dans une chaîne. Et un mot mal orthographié est refusé avec le mot le plus proche. C'est « un lexique, deux directions » en trois lignes :

HARO

```
try
        Q(5).VowelNB()
catch
        ? cCatchError
done
#--> NNL: a stznumber cannot count 'numberofvowels' -- and neither can the chain subject.
? Q("haro").IsAQ(:String).IfSoQ("majuscule").Content()
#--> HARO
try
```

```
        Q("haro").IsAQ(:String).IfSo("uppercse")
catch
        ? cCatchError
done
#--> NNL: this stzstring does not understand 'uppercse'. Did you mean 'uppercase'?
```

✓ exécuté : a affiché ce que ce bloc promet

Une chaîne se ramifie sur sa propre vérité, et un ordinal est un mot :

HARO

```
? Q("haro").IsAQ(:String).IfSoQ(:Uppercase).OtherwiseQ(:Reverse).Content()
#--> HARO
? Q("hello").IsAQ(:Number).IfSoQ(:Reverse).OtherwiseQ(:Uppercase).Content()
#--> HELLO
? Q("hello brave world").TheNth(2, :Words)
#--> brave
```

✓ exécuté : a affiché ce que ce bloc promet

Les questions fonctionnent de la même manière. Un mot de force, le premier mot de la question (`What`, `HowMany` et `Is` dans le bloc ci-dessous), ouvre un cadre, un mot et une méthode. Une question de confirmation se termine par « or not ». Que garde la session quand l'une répond oui et l'autre non ? Elle garde chaque réponse et les replie, comme une narration :

HARO

```
? @@( WhatQ().TheQ().VowelsQ().OfQ("AnnIE").Is() )
#--> [ "A", "I", "E" ]
? HowManyQ().WordsQ().Of("hello brave world")
#--> 3
ClearAnswers()
? IsQ().TheQ().LengthQ().OfQ("Haro").TheSameAsQ().TheQ().LengthQ().OfQ("Ruby").OrNot()
#--> 1
? Why()
#--> yes: 4 is the same as 4
? IsQ().TheQ().LengthQ().OfQ("Haro").TheSameAsQ().TheQ().LengthQ().OfQ("softanza").OrNot()
#--> 0
? Why()
#--> no: 4 is not the same as 8
? @@( AnswersSoFar() )
#--> [ 1, 0 ]
? AllYesSoFar()
```

```
#--> FALSE
? AnyYesSoFar()
#--> TRUE
```

✓ exécuté : a affiché ce que ce bloc promet

La session a gardé `[ 1, 0 ]` et l'a repliée : tout oui vaut `FALSE`, au moins un oui vaut `TRUE`.

LA MACHINE EST L'ANALYSEUR

Maintenant l'autre direction. La phrase est une chaîne de caractères, et la bibliothèque la lit. Le même verdict peut venir d'une narration de questions et d'une chaîne de vérité. Avec quelle certitude la chaîne de vérité répond-elle, et pourquoi ? Exécutez-la :

HARO

```
oT = Naturally("Create a string with 'haro' Is it lowercase ? Does it contain 'r' ?")
? @@( oT.Answers() )
#--> [ 1, 1 ]
? oT.AllYes()
#--> TRUE
? oT.Understood()
#--> create a string with haro -> ask: is lowercase -> ask: contains
TruthOf("haro") {
        IsA(:Lowercase)
        AndIsA(:Latin)
        AndContaining("r")
        ? Holds()
        #--> TRUE
}
? Why()
#--> yes: "haro" is lowercase and is latin and containing "r"
? Certainty()
#--> 1
? Evidentially()
#--> certainly
```

✓ exécuté : a affiché ce que ce bloc promet

Elle répond `certainly`, et dit pourquoi : `yes: "haro" is lowercase and is latin and containing "r"`.

Le paragraphe entier est compilé en code avant que rien ne s'exécute, et le code est lisible. C'est le compte rendu même de la bibliothèque sur la raison d'être de la couche :

HARO

```
o1 = Naturally("Create a string with 'test.data' Replace '.' with '_' Uppercase it")
? o1.Result()
#--> TEST_DATA
? o1.Code()
#--> oStr = StzStringQ("test.data")
#    oStr.Replace(".", "_")
#    oStr.Uppercase()
#    @result = oStr.Content()
```

✓ exécuté : a affiché ce que ce bloc promet

Le même sens en haoussa aboutit-il au même code ? Le haoussa est écrit ici sans la lettre crochetée, comme le paquet la lit aujourd'hui, pour une raison que donne le chapitre 11. Exécutez, et chaque programme redit ce qu'il a compris, dans ses propres mots :

HARO

```
oE = Naturally("Make a string with 'hello niger' inside Spacify it and uppercase it")
oH = NaturallyIn("hausa", "Yi rubutu da dauke 'hello niger' a ciki Raba shi kuma maida shi")
? oE.Code() = oH.Code()
#--> TRUE
? oH.Understood()
#--> yi rubutu dauke hello niger -> raba -> maida
? oE.Understood()
#--> create a string with hello niger -> spacify -> uppercase
```

✓ exécuté : a affiché ce que ce bloc promet

La comparaison répond `TRUE` : un seul code pour deux langues !

Le dictionnaire de phrases n'est pas le seul vocabulaire. Où va un mot qu'il ne tient pas ? Il est cherché dans le vocabulaire propre de la bibliothèque, ici les balises de documentation du verbe de mise en majuscules. Une valeur entre guillemets est une donnée, et jamais un verbe. Un mot inconnu en position de valeur reste un littéral :

HARO

```
? Naturally("Create a string with 'softanza' Capitals it").Result()
#--> SOFTANZA
? Naturally("Create a string with 'capitals' Uppercase it").Result()
#--> CAPITALS
? Naturally("Create a string with 'a-sep-b' Replace sep with '_'").Result()
#--> a-_-b
```

✓ exécuté : a affiché ce que ce bloc promet

La troisième ligne est aussi une limite : elle a imprimé `a-_-b`. Le mot `sep` n'était dans aucun dictionnaire, et il a servi de texte à remplacer. Un mot qu'elle ne connaît pas est un littéral.

Cinq langues comme données. Le dictionnaire est indexé par sens, et le mot de chaque langue aboutit à un seul identifiant. Un mot français, un mot arabe et un mot turc aboutissent-ils au même ? Exécutez :

HARO

```
? StzResolveSemanticInLang("fr", "doublons")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemanticInLang("ar", "تكرارات")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemanticInLang("tr", "tekrarları")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemantic("removeduplicates")
#--> METHOD_REMOVEDUPLICATES
```

✓ exécuté : a affiché ce que ce bloc promet

Les langues suivent leur propre grammaire, et ce ne sont pas des traductions mot à mot. Le turc met le verbe en dernier, avec les paramètres avant lui. L'arabe, avec ses marques de voyelles et sa lettre doublée, est dépouillé par le paquet ; le suffixe du verbe est le « le » masculin pour un texte. Le français élide :

HARO

```
? NaturallyIn("tr", "'a.b' ile bir metin oluştur '.' yerine '_' koy").Result()
#--> a_b
? NaturallyIn("tr", "'haro' ile bir metin oluştur ve tersine çevir").Result()
```

```
#--> orah
? NaturallyIn("ar", "أنشئ نصا بـــ 'haro' اقلبه ثم").Result()
#--> orah
? NaturallyIn("ar", "أنشئ نصًا بـــ 'haro' اقلِبْه ثمّ").Result()
#--> orah
? NaturallyIn("ar", "أنشئ نصا بـــ 'haro' اقلبه ثم").Understood()
#--> أنشئ سلسلة بـــ haro -> اعكس
? NaturallyIn("fr", "Crée une chaine avec 'haro' et mets la à l'envers").Result()
#--> orah
```

✓ exécuté : a affiché ce que ce bloc promet

Trois langues, trois grammaires, une seule réponse : `orah`. L'arabe redit ce qu'il a compris, dans ses propres mots : `أنشئ سلسلة بـــ haro -> اعكس`.

Aucun lecteur natif n'a lu ces phrases. Elles sont calquées sur celles de la garde, avec « haro » à la place du mot d'essai de la garde, et l'inversion turque d'une chaîne de caractères est la variante du livre de l'inversion de liste de la garde. Je ne sais pas qu'un locuteur les ait lues. Le dossier ne tient aucune lecture, par un locuteur, des phrases arabes, turques, françaises ou swahilies, comme il n'en tient aucune pour le haoussa du chapitre 11. Elles sont imprimées comme des phrases calquées sur celles de la bibliothèque, non lues.

Une langue peut-elle être un bloc de données ? Voici le swahili, ajouté à l'exécution, dans les mots propres de la garde :

HARO

```
StzAddNaturalLanguage([
        :code = "sw",
        :name = "swahili",
        :script = "latin",
        :ignored_words = [ "na", "ya", "kwa", "hii" ],
        :semantic_mappings = [
                [:natural = "unda",   :semantic = "CREATE_OBJECT"],
                [:natural = "orodha", :semantic = "OBJECT_LIST"],
                [:natural = "na",     :semantic = "VALUE_INDICATOR"],
                [:natural = "panga",  :semantic = "METHOD_SORT"]
        ],
        :phrases = [
                [:semantic = "METHOD_REMOVEDUPLICATES", :words = "ondoa nakala, futa nakala"]
        ]
])
? StzHasLanguagePack("sw")
#--> TRUE
```

```
osw = NaturallyIn("sw", "Unda orodha na [ 9, 4, 9 ] ondoa nakala")
? @@( osw.Result() )
#--> [ 9, 4 ]
? osw.Understood()
#--> unda orodha na [ 9, 4, 9 ] -> ondoa nakala
```

✓ exécuté : a affiché ce que ce bloc promet

Quatre correspondances, une locution, et la sixième langue répond `[ 9, 4 ]` !

Le code naturel est aussi une donnée. Les blocs délimités d'un document s'exécutent, et un gabarit se remplit :

HARO

```
cDoc = "This tutorial explains deduplication." + nl +
        "#<" + nl +
        "Create a list with [ 3, 1, 3 ] and remove its duplicates" + nl +
        "#>" + nl +
        "The end."
? len( StzNaturalBlocks(cDoc) )
#--> 1
? @@( NaturallyFromDoc(cDoc)[1].Result() )
#--> [ 3, 1 ]
cT = "Create a list with {#1} and remove its duplicates"
? @@( NaturallyWith(cT, [ [ 5, 3, 5 ] ]).Result() )
#--> [ 5, 3 ]
? @@( NaturallyWith(cT, [ [ 9, 9, 4 ] ]).Result() )
#--> [ 9, 4 ]
```

✓ exécuté : a affiché ce que ce bloc promet

Ce que les paquets savent dire. Combien chaque paquet sait-il dire ? La bibliothèque a une fonction pour cela, et aucune garde n'imprime ses chiffres. Je compte par rapport à vingt-cinq sens de référence, qui sont les sens structurels et chaque méthode et modificateur que nomme un paquet livré. Ce ne sont pas les milliers de verbes que la bibliothèque a fait croître :

HARO

```
for cLang in [ "en", "ha", "fr", "ar", "tr" ]
        aC = StzPackCoverage(cLang)
        ? cLang + " " + aC[:covered] + " of " + aC[:total]
next
```

```
#--> en 23 of 25
#    ha 9 of 25
#    fr 25 of 25
#    ar 16 of 25
#    tr 10 of 25
? StzHasNeuralModel()
#--> FALSE
```

✓ exécuté : a affiché ce que ce bloc promet

Le paquet haoussa parle neuf des vingt-cinq sens : `ha 9 of 25` ! Et aucun modèle n'est chargé. La couche a un secours facultatif par plongement, quand rien de lexical n'a été trouvé et qu'un modèle est chargé. À ce commit aucun ne l'est, et aucun n'est nécessaire. Je dis donc « aucun modèle n'est chargé », et non « ne touche jamais un modèle ».

DU SCHÉMA DE 2020 AUX TROIS ÈRES

La bibliothèque nomme trois ères, dans un document de conception du 10 juillet 2026. La première est le langage quasi naturel, le premier paradigme. La deuxième est la programmation naturelle : abaisser le langage vers le code, avec du texte libre, et un objet par bloc. La troisième, de 2026, est un lexique partagé par l'exécution et la documentation, des paquets multilingues avec morphologie, et `Understood()`. J'appelle la troisième *le lexique partagé*. Le titre même de la vision pour elle emploie un mot que ce livre n'emploie pas.

Voici les dates, telles que le dossier les donne. Mon schéma stratégique porte les années 2020 à 2025, partagé le 13 septembre 2026 ; sa bande Narrations se lit « langage naturel exécutable ». Le diaporama de 2022 a été exporté le 20 septembre 2022, et sa diapositive 2 dit d'exprimer la pensée dans un langage naturel exécutable. Le dépôt de la bibliothèque commence le 12 mars 2022, et son historique ne peut pas dater 2020 ni 2021. Les fichiers quasi naturels et le moteur à son chemin actuel ont été commités pour la première fois entre le 16 et le 19 septembre 2025. Le 30 octobre 2025, les langues sont devenues de la configuration. Le 2 novembre 2025, l'entrée par chaîne de caractères a remplacé la capture par accolades. Le 9 juillet 2026, le lexique a été unifié à partir des balises de documentation. Le 10 juillet 2026, les deux documents de conception ont été écrits, et l'audit de quinze exemples

canoniques a trouvé que quatre tournaient, et que deux d'entre eux répondaient correctement. La page phare du site est du 8 octobre 2026. Les dates git viennent du dépôt propre de la bibliothèque, que l'instantané n'a pas ; je les y ai vérifiées le 9 octobre 2026, et les commits cités sont des ancêtres de l'épinglage.

La plus ancienne diapositive que le dossier date est donc la diapositive fondatrice du diaporama de 2022, exporté le 20 septembre 2022. L'année 2020 est la première année de mon schéma stratégique, partagé le 13 septembre 2026 ; le dossier ne tient aucune diapositive de 2020.

Vient maintenant la décision, et ce que j'ai refusé.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Je l'ai construite sans modèle d'IA, comme un dictionnaire. La conception dit que tout ce qui est dans le dossier naturel attaque un seul mur : l'écart sémantique entre une intention humaine et son exécution. La thèse du langage quasi naturel, dans cette revue : zéro NLP, zéro ML, vitesse native, déterminisme complet, et le programmeur aux commandes. L'observation finale de la vision naturelle est que la couche a livré une quatrième promesse que les anciens documents n'avaient jamais osé faire : la responsabilité.

J'ai décidé de deux directions, une seule discipline. Le tableau de la revue a cinq lignes. Qui analyse : une machine avec un lexique, ou l'humain. L'ambiguïté : refusée bruyamment, ou impossible par construction. Le vocabulaire : borné par les paquets, ou non borné. La compréhension : vérifiée par la machine, ou par une lecture humaine. Le public : des experts du domaine et des agents d'IA, ou des programmeurs. Les deux finissent au même endroit : une suite déterministe d'opérations sémantiques.

J'ai refusé les marqueurs de référence. Le 10 juillet 2026, j'ai rejeté la famille de suffixes qui marquaient une référence. Le chiffre a été inversé par son propre inventeur, de mémoire, il dupliquait des sens, et le langage lui-même ne marque jamais la référence ainsi. Plus de deux cent soixante méthodes ont été retirées. Le chapitre 11 le raconte en deux phrases.

J'ai refusé la chaîne non grammaticale. La chaîne est la phrase, et une chaîne non grammaticale qui s'exécute est un bogue. Je tiens cela pour une vue, non pour un résultat. La doctrine est tenue par la lecture à voix haute, non par une machine, et j'en montre la preuve plus bas.

J'ai refusé la négation en préfixe. La négation vit dans les formes négatives, dans les comparateurs antonymes et dans le fermeur de question `OrNot()`. La question de confirmation ci-dessus est cette décision en train de tourner.

J'ai refusé la tolérance silencieuse. L'ancienne tolérance aux fautes de frappe a été remplacée par l'explication de tout : le refus avec le mot le plus proche, ci-dessus.

J'ai refusé le contexte global. Deux chaînes entremêlées se contaminaient, et la correction a mis le contexte sur la chaîne. Le `Why()` à l'échelle du processus reste comme commodité pour la console et les tests. Les globales survivent comme adaptateurs de repli : une ancienne méthode hors de la couche naturelle qui construit un nouvel objet y retombe encore.

J'ai refusé le langage de balisage. Les annotations qui donnaient à la machine ce qu'elle ne pouvait pas inférer gouvernaient en obligeant l'humain à annoter. Le système gouverne par le dialogue. Deux graines ont été gardées comme portes minces, et ce sont le document et le gabarit ci-dessus.

J'ai refusé un interpréteur de commandes pour des récits, et la capture par accolades. Dans mes propres mots dans l'article : il est conçu pour des commandes, pas pour des récits. Et un objet par bloc. La forme chaîne de caractères a remplacé la forme à accolades, parce que le code naturel est devenu un type de données de première classe, stockable, versionnable et cherchable. La règle depuis assouplie : un programme peut nommer des objets et passer de l'un à l'autre.

J'ai refusé la gouvernance à l'intérieur de la couche naturelle. J'ai sorti cette portée de la bibliothèque le 12 juillet 2026, et le code a été retiré.

J'ai refusé le langage naturel comme moyen de déclarer. Il demande, et ne déclare jamais.

Le tribunal vient ensuite, et l'une de ses gardes échoue.

5. LE TRIBUNAL

`nnl_narrated` prouve la phrase phare, la surface générée de noms, les mots de degré, le mode conditionnel, la référence ordinale, le refus qui rend compte de lui-même, la doctrine de grammaticalité, le contexte sur la chaîne, et le temps du discours. `multilingual_pack_narrated` prouve les paquets comme données, des

programmes en français, en arabe et en turc qui s'exécutent, le retrait d'affixes vérifié, une langue ajoutée à l'exécution, et `Understood()` rendu dans la langue du programme. `question_narrated` prouve les cadres de question et qu'un cadre sans aspect est refusé. `literate_templates_narrated`, les documents et les gabarits. `truthchain_narrated` et `interrogative_narrated`, les verdicts repliés.

`semantic_resolver_narrated` est la garde qui devrait protéger le dictionnaire, et à ce commit elle échoue. Je l'ai lancée deux fois le 8 octobre 2026 et, chaque fois, elle s'est arrêtée sur une erreur : trois assertions ont échoué, et l'exécution s'est terminée après la vingt-neuvième. Dans cette exécution, le mot « caps », que la garde affirme, s'est résolu en une méthode pour les bornes, et « mirrored » ne s'est résolu en rien. La raison est celle de la conception elle-même : le vocabulaire est lu dans les balises de documentation de la bibliothèque, plus dix-sept opérations écrites à la main, portées par réflexion à environ trois mille. Une modification de la documentation change donc le sens d'un mot. La bibliothèque refuse honnêtement dans le second cas : dans cette exécution, `Unresolved()` a nommé le mot et `UnderstoodAll()` a imprimé 0. Le tribunal du dictionnaire a échoué sur le dictionnaire, et le chapitre 11, le chapitre 22 et le chapitre 2 nomment cette garde.

J'ai lancé une fois chacune les autres gardes nommées ci-dessus le 8 octobre 2026, et dans cette exécution elles ont passé toutes leurs assertions ; les données propres du livre n'en tiennent que les décomptes. Les appels d'assertion écrits dans les sept gardes nommées ci-dessus totalisent 269.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`nnl_narrated` 87 assertions base/test/natural/nnl_narrated.ring

`multilingual_pack_narrated` 45 assertions
base/test/natural/multilingual_pack_narrated.ring

`semantic_resolver_narrated` 83 assertions
base/test/natural/semantic_resolver_narrated.ring

`question_narrated` 21 assertions
base/test/natural/question_narrated.ring

`literate_templates_narrated` 10 assertions
base/test/natural/literate_templates_narrated.ring

`truthchain_narrated` 12 assertions
base/test/natural/truthchain_narrated.ring

`interrogative_narrated` 11 assertions
base/test/natural/interrogative_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en largeur de formulation. Un modèle d'IA accepte n'importe quelle formulation dans n'importe quelle langue, et le dictionnaire accepte ce que les paquets nomment. Mesuré plus haut : le paquet haoussa couvre neuf des vingt-cinq sens, le turc dix, l'arabe seize, l'anglais vingt-trois (les deux qui lui manquent, l'anglais les atteint par le vocabulaire développé par réflexion), le français vingt-cinq.

Et la couche de phrases n'agit que sur trois classes : les chaînes de caractères, les listes et les nombres. Les objets du domaine d'un client n'en font pas partie ; le domaine passe par le graphe de connaissance.

Le terrain mène en sémantique de programme entier. Inform 7 compile un monde entier. Ici un programme est une suite d'opérations sur un objet vivant.

Le terrain mène sur l'éditeur. Dans mon exécution, la fonction de suggestion a répondu neuf suites possibles pour une phrase à moitié écrite, et le code attribue au terrain des langages contrôlés l'idée. Il n'y a pas ici d'éditeur qui les montre pendant que vous tapez.

Et il y a une perte du côté de la bibliothèque elle-même. `Understood()` imprime les verbes et non leurs arguments : le bloc de la section 3 imprime `contains` et non le « r » demandé. Dans mon exécution, `Replace` s'est imprimé `replace`, et `Add 3 to it` s'est imprimé `add` alors que le résultat était 8. Un lecteur qui vérifie la lecture de la machine ne voit pas de 3. Cela nuance l'affirmation du chapitre 11 selon laquelle une personne peut vérifier ce que la machine a compris.

Là où est le gain, avec la perte à côté : la même phrase donne la même réponse, avec une raison, dans la langue de l'utilisateur, et conçue pour s'exécuter hors ligne, ce qu'aucune garde ne mesure. La perte est que la langue de l'utilisateur doit être dans un paquet, et qu'un mot extérieur est un littéral ou un refus. Le dossier dit que le cas gagnant est le travail réglementé ou souverain : une instruction qui s'exécute localement et peut être auditée au mot près.

La section suivante dit où le chapitre s'arrête.

7. LA LIMITE

CONSTRUIT — les deux directions : des chaînes où l'humain est l'analyseur, et des phrases où c'est la machine ; cinq paquets de langue comme données et un sixième ajouté à l'exécution ; des refus avec raisons des deux côtés ; des cadres de question ; une chaîne de vérité mince ; du code naturel comme données dans les documents et les gabarits

CONÇU — une grammaire qui tient un modèle d'IA au lexique ; une garde qui vérifie qu'aucun socket ne s'ouvre pendant une exécution naturelle ; le plan émis

comme une chaîne ; les exemples des articles épinglés comme fixtures

ABSENT — toute machine qui juge la grammaire d'une chaîne ; l'ajami, le zarma et le peul ; une couche naturelle fermée jusqu'en bas ; un dictionnaire protégé de la dérive de la documentation dont il est lu ; les promesses des anciens articles, dont plusieurs ne tournent pas

Une chaîne non grammaticale s'exécute. La seconde chaîne ci-dessous se lit « which a length of 4 », ce qui n'est pas de l'anglais. Quelque chose la juge-t-il ? Exécutez les deux :

HARO

```
? TheWordQ("haro").IsAQ(:Word).WhichQ().HasQ().ALengthQ().Of(4)
#--> TRUE
? TheWordQ("haro").IsAQ(:Word).WhichQ().ALengthQ().Of(4)
#--> TRUE
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux répondent `TRUE` : rien n'a jugé la seconde ! La doctrine est que relire une chaîne, c'est la lire à voix haute comme de l'anglais. C'est le travail de l'humain.

Le descripteur d'une chaîne atteint-il le code hôte ? Oui. `IsAXT` construit une chaîne de caractères et l'évalue, de sorte qu'une fonction définie par le programme et nommée dans la chaîne a imprimé `host code ran`. C'est le résidu que l'étude du dossier et le chapitre 6 nomment :

HARO

```
? Q("haro").IsAXT([ "String('x') and HostRan() and @isString" ])
#--> host code ran
#    TRUE

func HostRan()
        ? "host code ran"
        return 1
```

✓ exécuté : a affiché ce que ce bloc promet

Et un mot de l'ancienne ère est signalé, et jamais deviné. Le mode permissif exécute quand même le reste, et le mode strict refuse. La suggestion offerte, `addn`, est mauvaise :

HARO

```
o = Naturally("Create a string with 'softanza' Spacify it and_ Uppercase it")
? o.Result()
#--> S O F T A N Z A
? o.UnderstoodAll()
#--> FALSE
? @@( o.Unresolved() )
#--> [ [ "and_", "addn" ] ]
try
        NaturallyStrict("Create a string with 'softanza' Spacify it and_ Uppercase it")
catch
        ? cCatchError
done
#--> Strict natural mode: not understood: 'and_' (did you mean 'addn'?)
```

✓ exécuté : a affiché ce que ce bloc promet

Plusieurs chaînes des anciens articles ne tournent plus. La phrase de l'article sur le pont qui porte `IsAMQ` demande un marqueur que j'ai retiré par ma décision du 10 juillet 2026. Plusieurs chaînes de l'article quasi naturel ne tournent pas à ce commit : `InLowercase()` lève une erreur faute d'objet ; `LettersNB`, `BeforeQ` et `TheirQ().FirstCharQ()` ne sont pas définis ; et la forme à accolades `Naturally() { ... }` lève une erreur pour trop peu de paramètres. La chaîne de l'article sur le pont avec `ReplaceLowercaseCharsQ` n'est pas définie non plus. J'ai lancé douze chaînes de l'article quasi naturel le 8 octobre 2026, et huit fonctionnent. Dans cette exécution, le paragraphe de l'article sur le pont tournait encore en mode permissif, et imprimait deux boîtes imbriquées autour de « NOTHING SOFTANZA » au lieu de la seule qu'il promet. La machine l'a dit, puisque `UnderstoodAll()` vaut faux et que le mot non résolu est nommé (le bloc ci-dessus montre les deux), mais un lecteur doit le demander.

Ce qui est construit : les deux directions ; cinq paquets comme données et un sixième à l'exécution ; des refus avec raisons des deux côtés ; les cadres de question, la chaîne de vérité mince, le registre évidentiel ; le code naturel comme données. Le

dossier contient treize fichiers et environ douze mille lignes dans douze classes, d'après `wc` et `grep` le 8 octobre 2026.

Ce qui est conçu : le plan d'une narration émis comme une chaîne exécutable, de sorte que les chaînes gagnent `Understood()` gratuitement. Une grammaire émise pour le lexique naturel, de sorte qu'un modèle d'IA ne puisse produire que des phrases que la couche exécute : j'ai cherché dans le dossier, et aucun émetteur de grammaire ne le touche. Une garde qui échoue si un socket s'ouvre pendant une exécution naturelle, qui n'existe pas. Un chiffre de couverture par paquet imprimé par une garde. Et les exemples des articles épinglés comme fixtures.

Ce qui est absent : le haoussa en écriture ajami, le zarma et le peul. Une couche naturelle fermée jusqu'en bas : une liste simple dans une phrase naturelle atteint le code hôte, ce que le chapitre 11 imprime, et un descripteur atteint le code hôte, ci-dessus. Toute machine qui juge la grammaire d'une chaîne. La lettre haoussa standard, que le paquet stocke brisée. Un dictionnaire protégé de la dérive. Le dossier ne tient pas non plus de récit, ni de date, de mon plaidoyer auprès du projet de l'ancien langage hôte pour que la programmation naturelle devienne une préoccupation première, ni une phrase de moi sur ce ton dont le livre puisse citer la source. Il ne tient qu'un mémo privé du 30 août 2026, qui donne mon jugement sur la forme naturelle livrée par ce langage. Le mémo est marqué privé : le livre ne le cite pas et ne m'attribue aucune expression tirée de lui.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Écrivez la chaîne comme la phrase et lisez-la à voix haute. La bibliothèque refuse ce que son sujet ne peut pas faire, et dit pourquoi. Vous êtes l'analyseur, donc la phrase n'est grammaticale que dans la mesure où vous l'êtes.
L'analyste	Une exigence peut être un paragraphe qui s'exécute, avec une paraphrase qui revient. Aujourd'hui elle agit sur des chaînes de caractères, des listes et des nombres, donc les mots de domaine de l'analyste vont dans un fichier de connaissance (chapitre 14), pas ici.
Le concepteur	Le ton d'un concepteur est dans les refus : un message nomme le mot et le mot le plus proche. Le dossier n'a pas de preuve propre au concepteur pour ce chapitre, et je garde la ligne courte.
Le décideur	Demandez à toute affirmation de « langage naturel » ce qu'elle fait d'un mot qu'elle ne connaît pas. Ici elle le signale, le refuse en mode strict, ou le prend comme littéral, et dit lequel. Elle est conçue pour s'exécuter hors ligne, sans clé. L'affirmation d'absence de réseau n'est pas encore mesurée.
Le formateur	Un apprenant écrit en haoussa, en français, en arabe ou en turc, et le programme redit ce qu'il a compris dans la même langue. Le paquet haoussa parle neuf des vingt-cinq sens, et aucun locuteur de ces langues n'a lu les phrases de cette page, pour autant que le dossier le montre.
L'agent	Parlez le lexique. Utilisez le mode strict et une liste d'autorisation (chapitre 11). Attendez-vous à un refus avec une raison. Votre phrase est une proposition. Une grammaire qui vous tiendrait au lexique est conçue et non construite.

9. EXERCICE

Prenez une phrase que vous dites d'un mot de votre propre langue : qu'il est en minuscules, qu'il a une longueur donnée, et un nombre donné de voyelles. Écrivez-la deux fois avant de rien exécuter. D'abord comme une chaîne quasi naturelle, où vous êtes l'analyseur. Ensuite comme une phrase `Naturally`, où c'est la bibliothèque. Sous chacune, écrivez la ligne que vous vous attendez à voir imprimée.

HARO

```
# 1. as a chain, where you are the parser:
# 2. as a Naturally sentence, where the library is:
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez ensuite les deux. Puis changez un mot de la chaîne, et un mot de la phrase, en un mot que le dictionnaire ne tient pas. Quelle direction vous l'a dit ? Laquelle avez-vous dû vous le dire vous-même ? Dans une direction la machine vous le dit, et dans l'autre vous devez vous le dire vous-même. Si une promesse se rompt, ne la changez pas : expliquez d'abord la sortie.

Je vous laisse celui-ci. La section 7 montre une chaîne qui se lit « which a length of 4 », et rien ne l'a refusée. Exécutez-la, lisez-la à voix haute comme de l'anglais, et écrivez dessous la ligne que vous voudriez voir la bibliothèque imprimer pour la refuser, avec la raison qu'elle donnerait.

HARO

```
# the chain of section 7 that nothing judged:
? TheWordQ("haro").IsAQ(:Word).WhichQ().ALengthQ().Of(4)
# the line you would have the library print to refuse it, with its reason:
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Quand vous aurez écrit cette ligne, vous saurez pourquoi je tiens la chaîne non grammaticale pour une vue et non pour un résultat, et qui la page dit qu'il faut pour la juger (sections 4 et 7).

Le chapitre suivant porte sur la programmation de la connaissance, où les mots de domaine de l'analyste ont été envoyés. Son principe est qu'un système est un fichier, et qu'y ajouter une ligne fait que le système sait davantage, sans rien coder ni entraîner. Ici une phrase demande, et ne déclare jamais.

OÙ SONT LES PREUVES

NARRATION	`stznatural-vs-ring-naturallib.md`
NARRATION	`stz-near-natural-language-programming.md`
NARRATION	`stznatural-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
SUR LE SITE	natural
DÉCISIONS DE L'ÉTUDE	12.2, 12.6, 12.27, 12.30

PARTIE III · CHAPITRE 14

La programmation de la connaissance

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 13 bloc(s) sur 13 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends les trois pratiques du terrain qui se rencontrent dans cette idée : la pile sémantique, la base de données de graphes de propriétés et les règles sur des faits. Je mets à côté d'elles un système qui est un fichier texte brut. J'exécute ce fichier : deux faits, une loi et une preuve avec ses étapes, puis une clinique en mode strict, où une rumeur refusée est gardée et nommée. J'écris le fichier, je le relis et je le montre à l'échelle, d'un catalogue de cours à la mémoire d'un agent d'IA. Je dis ce que j'ai décidé, qu'une entreprise se décrit dans un seul fichier et que le fichier est le programme. Je dis ce que j'ai refusé : un monde parallèle, une contradiction résolue en silence, la récupération. J'imprime la limite, un fichier qui à ce commit n'a pas de règles. Elle se tient à côté du principe : un système est un fichier, et y ajouter une ligne fait que le système sait davantage, sans rien coder ni entraîner.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Trois pratiques du terrain se rencontrent dans cette idée, et aucune n'est la même qu'elle.

La première est la pile sémantique : RDF, OWL et SPARQL. Un fait est un triplet de sujet, de prédicat et d'objet, écrit dans un vocabulaire partagé. Une ontologie dit quelles classes et quelles propriétés existent et ce qui en découle, et un raisonneur en dérive les conséquences. Ce sont des recommandations du W3C, de 2014, 2009 et

2013. Ce que le terrain demande à l'auteur d'une base de connaissance, c'est un langage d'ontologie à lui, un magasin et un langage de requête. Ils se tiennent d'ordinaire à côté de l'application et non dedans.

La deuxième est la base de données de graphes de propriétés, avec son langage de requête. Le dossier a une comparaison qui met de telles bases face à cette bibliothèque. C'est un texte qu'un modèle d'IA a écrit, gardé dans le dossier de la bibliothèque, et il reconnaît au terrain l'échelle, la persistance et les transactions.

La troisième est les règles sur des faits, en Prolog et en Datalog. Des faits et des règles dans un seul programme, une requête par résolution. Le dossier ne nomme que Prolog, comme un langage à rejoindre, et non à reconstruire.

Une quatrième pratique se tient derrière le contraste, et le document de conception ne la nomme que pour la refuser. C'est la récupération, un tas de fragments de documents cherchés par similarité ; cette description est une connaissance générale, pas une étude du dossier.

Ainsi, sur le terrain, la connaissance vit dans un magasin et sa logique dans un raisonneur, tous deux à côté du programme. Ici la connaissance est un fichier texte brut que le programme charge.

2. LE PREMIER PRINCIPE

Un système est un fichier. Y ajouter une ligne fait que le système sait davantage, sans rien coder ni entraîner.

Deux phrases du dossier le portent. La définition, dans le document de conception écrit à partir de ma directive du 13 juillet 2026 : l'intelligence est la capacité de calcul de dériver une connaissance nouvelle à partir d'une connaissance existante. Et ce que la vision de 2026 en garde : ajouter un fait augmente l'intelligence du système, sans changement de code et sans entraînement. Le chapitre 17 s'ouvre sur la première, et je m'ouvre ici sur la seconde. Le document de conception du 13 juillet 2026 le dit déjà, presque dans les mêmes mots, et le chapitre garde son ouverture.

Voyons maintenant la forme elle-même, un texte et ce qu'il fait quand on l'exécute.

3. LA FORME SOFTANZA

Voici un système comme un texte. Deux faits : la farine est faite de mil, et le couscous de farine. Le couscous est-il fait de mil ? Demandez-le au graphe, puis ajoutez une ligne au fichier, une loi disant que *fait de* est transitif, et demandez de nouveau :

HARO

```
cFacts = 'knowledge "coop"' + nl + nl + "facts" + nl +
         "    flour | made-from | millet" + nl + "    couscous | made-from | flour" + nl
oA = new stzKnowledgeGraph("coop")
oA.ImportKnow(cFacts)
? oA.Prove([ "couscous", "made-from", "millet" ])[:narration]
#--> not derivable: no recorded fact or lawful chain gives 'couscous' made-from 'millet'
oB = new stzKnowledgeGraph("coop")
oB.ImportKnow(cFacts + nl + "ontology" + nl + "    made-from | transitive" + nl)
? oB.Prove([ "couscous", "made-from", "millet" ])[:narration]
#--> proved: couscous made-from flour made-from millet
```

✓ exécuté : a affiché ce que ce bloc promet

Sans la loi, le graphe répond `not derivable` ; avec elle, `proved: couscous made-from flour made-from millet`. Le texte est le programme. Les deux mêmes faits, une ligne en plus, une capacité différente. Le bloc ne touche pas le disque : `ImportKnow` prend un texte, et prend un chemin quand la chaîne de caractères finit par l'extension du fichier.

Maintenant ajoutons un fait au graphe qui a la loi, et rien d'autre. Que répondent les deux graphes à la question plus longue ? Exécutez-le :

HARO

```
oB.KnowRelation("porridge", "made-from", "couscous")
? oB.Prove([ "porridge", "made-from", "millet" ])[:narration]
#--> proved: porridge made-from couscous made-from flour made-from millet
? oA.Prove([ "porridge", "made-from", "millet" ])[:narration]
#--> not derivable: no recorded fact or lawful chain gives 'porridge' made-from 'millet'
```

✓ exécuté : a affiché ce que ce bloc promet

Le graphe qui a la loi prouve la chaîne plus longue et en écrit les étapes : `proved: porridge made-from couscous made-from flour made-from millet`. Le graphe sans la loi, qui n'a de plus jamais reçu le fait, répond toujours `not derivable: no recorded fact or lawful chain gives 'porridge' made-from 'millet'`, et je veux que vous remarquiez que ce non a deux causes, nommées dans sa propre ligne : aucun fait enregistré, et aucune chaîne que la loi permette. Un refus est aussi certain qu'une preuve, sur ce qui est enregistré et déclaré : le graphe dit « non dérivable », pas « faux ». Le commentaire de la bibliothèque sur `Prove` le dit : `a certain no is still certain`.

La même croissance atteint le modèle de langage de domaine du chapitre 19. Rien n'est entraîné. Mais un modèle forgé à partir d'un graphe, dans le mot du chapitre 19, est un instantané. Le modèle apprend-il le fait nouveau quand le graphe l'apprend ? Exécutez-le :

HARO

```
oKB = new stzKnowledgeGraph("coop")
oKB.Know("couscous", "dish")
oKB.KnowRelation("couscous", "contains", "semolina")
oDLM = StzDlmQ(oKB)
? oDLM.Ask("what is millet?")
#--> That is outside the 'coop' domain (I know 1 entities and 1 relations).
oKB.Know("millet", "grain")
oKB.KnowRelation("couscous", "contains", "millet")
? oDLM.Ask("what is millet?")
#--> That is outside the 'coop' domain (I know 1 entities and 1 relations).
oDLM = StzDlmQ(oKB)
? oDLM.Ask("what is millet?")
#--> Millet is a grain.
? oDLM.Ask("couscous contains what?")
#--> Couscous contains semolina and millet.
```

✓ exécuté : a affiché ce que ce bloc promet

Il ne l'apprend pas. La seconde réponse est l'ancien modèle qui refuse encore, `That is outside the 'coop' domain`, et seul le modèle forgé de nouveau répond `Millet is a grain.`

Le fichier que la bibliothèque écrit. Une clinique. Une relation déclarée unique, et un graphe en mode strict, où chaque fait a besoin de sa provenance. Que fait-il d'un fait sans source, d'un fait qui en a une, et d'un second, qui le contredit, venu d'une rumeur ? Et qu'écrit-il ensuite sur le disque ? Exécutez-le :

HARO

```
StzKnowledgeGraphQ("clinic") {
        ConstrainRelation("treated-by", :Unique)
        EnableStrictMode()
        try
                AddFact("fatou", "treated-by", "dr-diallo")
        catch
                ? cCatchError
        done
        #--> Strict mode: every fact needs provenance -- use AddFactXT(s, p, o, [ :source = ...,
:confidence = ... ]).
        AddFactXT("fatou", "treated-by", "dr-diallo", [ :source = "registry", :confidence = 0.95
])
        ? AddFactXT("fatou", "treated-by", "dr-sani", [ :source = "rumour", :confidence = 0.2 ])
        #--> 0
        WriteToKnowFile("clinic")
}
? read("clinic.zknw")
#--> knowledge "clinic"
#
#    facts
#        fatou | treated-by | dr-diallo
#
#    provenance
#        fatou | treated-by | dr-diallo | source | text | registry
#        fatou | treated-by | dr-diallo | confidence | number | 0.95
#
#    contradictions
#        fatou | treated-by | dr-diallo | dr-sani | rumour
#
#    ontology
#        treated-by | unique
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le fait nu est refusé avec la phrase propre de la bibliothèque, `Strict mode: every fact needs provenance`. Le fait avec sa source et sa confiance est admis. La rumeur est refusée, `0`, et le refus est gardé ! C'est ce que la bibliothèque écrit, un en-

tête et jusqu'à quatre sections. Elles viennent dans cet ordre : les faits, la provenance quand un fait en porte, les contradictions quand l'une a été refusée, et l'ontologie quand une loi a été déclarée. Une ligne par enregistrement, les champs séparés par une barre. Une barre ou un saut de ligne à l'intérieur d'une valeur est échappé. La rumeur refusée est dans le fichier, nommée, et jamais résolue. Le commentaire du code sur la liste des contradictions le dit : `named, NEVER silently resolved`.

Le fichier se relit-il dans un graphe strict, avec la loi, la source de chaque fait et le refus à l'intérieur ? Chargez-le :

HARO

```
StzKnowledgeGraphQ("clinic") {
        EnableStrictMode()
        aR = ImportKnow("clinic.zknw")
        ? aR[:merged]
        #--> 1
        ? @@( MetaOfFact("fatou", "treated-by", "dr-diallo") )
        #--> [ [ "source", "registry" ], [ "confidence", 0.95 ] ]
        ? len(Contradictions())
        #--> 1
        ? RelationHasLaw("treated-by", "unique")
        #--> 1
}
remove("clinic.zknw")
? fexists("clinic.zknw")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Oui : un fait fusionné, la source et la confiance revenues, `[ [ "source", "registry" ], [ "confidence", 0.95 ] ]`, une contradiction, et la loi. Un fichier de ce genre a été écrit un jour sans sa provenance. L'export n'écrivait que les triplets nus, de sorte que la connaissance relue depuis le disque ne disait plus d'où elle venait. Un graphe strict ne pouvait pas charger son propre export. La correction, ce sont les deux sections ci-dessus. La garde qui la prouve est nommée à la section 5.

Une loi qui se déclenche quand un fait est ajouté. Sur le monde par défaut de la bibliothèque, celui qui est derrière le sucre syntaxique du langage naturel, une loi symétrique dérive le fait inverse au moment où le premier est ajouté, sans appel à

une étape de dérivation. Personne n'a écrit qu'Ibrahim est marié à Amina. Le monde par défaut le sait-il ?

HARO

```
StzConstrainRelation("married-to", :Symmetric)
StzKnowRelation("amina", "married-to", "ibrahim")
? DefaultKnowledgeGraph().Query([ "ibrahim", "married-to", "amina" ])
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `1`. Le fait inverse est là, et personne ne l'a écrit ! C'est le seul endroit où une loi déclarée se déclenche à l'ajout qu'une exécution de ce livre montre, et c'est le monde par défaut. La classe de graphe a aussi un interrupteur, `EnableAutoDerive`, qui lance les règles de dérivation après chaque arête ajoutée ; il est éteint par défaut, et je ne l'ai pas exécuté. Je montre la même loi sur un graphe d'instance à la section 7, où cela ne fonctionne pas.

Un système à l'échelle. Le programme d'éducation de la bibliothèque est un fichier de connaissance. Son catalogue, `program.zknw`, dit comment s'appelle le programme et quels cours et quelles langues il a. Quel est son nom, et quelles langues porte-t-il ? Demandez-le-lui :

HARO

```
oP = StzProgramQ("../../education/program")
? oP.Id()
#--> softanza-education
? @@( oP.Languages() )
#--> [ "en", "fr", "ar", "ha" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Il s'appelle `softanza-education`, et il a quatre langues. Ajouter un cours, c'est y ajouter une ligne, et la garde vérifie que chaque cours déclaré est un dossier et que chaque dossier est déclaré. Dans le module d'éducation, l'instantané tient quatre-vingt-dix-neuf fichiers de connaissance, comptés avec `find` le 8 octobre 2026. Ce sont des cours, des exercices, des mondes et des compétences, un monde de projet

pour un apprenant, et les surcouches qu'une institution pose sur le noyau. La mémoire d'un agent est aussi un graphe de connaissance, sauvegardé par le même format, avec la provenance.

Voilà la forme. Voyons maintenant ce que j'ai décidé, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé qu'une entreprise se décrit dans un seul fichier, et que le fichier est le programme. Le document de conception enregistre ma directive : des entités, des relations, une ontologie de sémantiques propres au domaine, et des règles, dans un fichier lisible, comparable et modifiable à la main, validé au chargement. Je n'ai pas encore rencontré les deux derniers mots, et la section 7 dit où.

J'ai décidé des extensions à quatre lettres. `.zknw` pour la connaissance, et les autres de la famille ; les anciennes graphies se lisent encore. L'instantané écrit `.zknw` et lit encore `.stzknow`. Le fichier de règles n'a pas été renommé : le graphe écrit et lit toujours `.stzrulz`, bien que la famille lui donne quatre lettres. Je le rapporte comme le dossier l'a.

J'ai refusé un monde parallèle. La couche naturelle tenait autrefois ses entités dans des globales, et le document de conception l'a appelé le mauvais foyer, la mauvaise forme et un monde parallèle. La résolution est un graphe par défaut, unique, derrière le sucre syntaxique du langage naturel. Les anciennes globales existent encore, et le graphe grandit à côté : l'en-tête de la bibliothèque appelle cela une double écriture. La loi qui le sous-tend est que les globales de domaine nues sont interdites ; une forme globale ne peut exister que comme mince sucre sur une instance par défaut. Ainsi le dialogue de la loi symétrique ci-dessus est du sucre sur un graphe de la même classe que celui des premiers blocs.

J'ai refusé de résoudre une contradiction en silence. Le mode strict, où chaque fait a besoin de sa provenance, est une hygiène de la connaissance, sur option.

J'ai refusé la récupération. Le document de conception dit : pas de changement de code, pas d'entraînement, pas de mécanique RAG. C'est le même refus que celui du chapitre 17.

5. LE TRIBUNAL

`knowledge_integration_narrated` prouve que le monde répond à `WhatIs`, que le graphe a grandi à côté, et que la loi vit dans l'ontologie. Elle prouve que `Prove` narre d'abord la loi puis les maillons de la chaîne, avec une certitude de 1, et refuse ce qu'il ne peut pas prouver. Elle prouve que le fichier fait l'aller-retour. Elle prouve qu'une loi symétrique dérive l'arête inverse quand l'étape de dérivation est appliquée après l'ajout d'un fait par le graphe. Enfin, elle prouve que le mode strict admet un fait muni de provenance, refuse un fait nu, et refuse et enregistre une contradiction d'unicité.

`provenance_survives_save_narrated` prouve que la source et la confiance d'un fait survivent à une sauvegarde, qu'un graphe strict charge son propre export, et qu'une contradiction refusée survit. Elle prouve qu'une barre ou un saut de ligne dans une valeur est une donnée, et qu'un ancien fichier sans provenance se charge encore. Elle prouve que la mémoire d'un agent enregistre qui a appris un fait, et quand.

`program_narrated` prouve que le programme compte ses cours, et que chaque cours déclaré est un dossier et chaque dossier est déclaré. Elle prouve que les textes se lisent de la même façon depuis un checkout avec d'autres fins de ligne.

Voilà ce que prouvent les gardes. Ce que le terrain fait mieux vient ensuite.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`knowledge_integration_narrated` 19 assertions
base/test/natural/knowledge_integration_narrated.ring

`provenance_survives_save_narrated` 23 assertions
base/test/security/provenance_survives_save_narrated.ring

`program_narrated` 21 assertions
base/test/education/program_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

La comparaison de graphes du dossier dit elle-même où les solutions traditionnelles gagnent : les systèmes à grande échelle qui ont besoin de stockage et de traitement distribués, et les applications où la performance est critique. Son tableau place Softanza à l'échelle de l'entreprise, de quelques centaines à quelques milliers de nœuds, dans un seul processus. C'est le résumé honnête de la perte : le terrain mène en échelle, en persistance, en transactions et en outillage. La ligne de la comparaison sur la persistance est antérieure au fichier ; le graphe de connaissance en écrit un maintenant.

Le terrain mène en normes et en portée, en connaissance générale. Une base de connaissance en RDF peut être partagée avec n'importe quel outil qui lit les formats du W3C, reliée à des vocabulaires publics et interrogée à travers des magasins. D'après le dossier, un fichier de connaissance est lu par cette bibliothèque et par rien d'autre. Un raisonneur pour un langage d'ontologie riche fait ce que trois lois et une déclaration de classe ne font pas. Le terrain mène en expressivité des règles : Prolog et Datalog dérivent des faits nouveaux à partir de règles arbitraires. Ici, dans les

exécutions que j'ai gardées, une seule sorte de loi dérive un fait, et seulement sur le monde par défaut. Le terrain mène en requêtes à plusieurs étapes, puisque SPARQL et Cypher joignent des motifs. Et en relations multiples entre deux choses.

Voici où je pense que cela se distingue, avec les limites mesurées à côté. La connaissance est un fichier texte qu'une personne peut lire, modifier, comparer et livrer avec le programme. Chaque réponse est prouvable avec ses étapes, et un non est aussi certain qu'un oui. Ce qu'un fait affirme voyage avec sa source et ses refus. Et le même format porte un cours, le projet d'un apprenant et la mémoire d'un agent. Rien de cela n'a besoin d'un magasin, d'un serveur, d'un modèle d'IA ni d'un entraînement.

Les pertes sont imprimées. La limite du chapitre vient ensuite, avec ses exécutions.

7. LA LIMITE

CONSTRUIT	un système comme un fichier texte brut de faits, avec des lois (unique, transitive, symétrique) et la source de chaque fait ; Prove avec ses étapes, un non aussi certain qu'un oui ; le mode strict ; le fichier écrit, relu et livré, d'un catalogue de cours à la mémoire d'un agent d'IA
CONÇU	des règles dans le fichier ; la dérivation sur n'importe quel graphe ; une requête qui joint ; plusieurs relations entre deux choses ; l'héritage de classes ; la validation au chargement
ABSENT	le dialogue fondateur de 2022, qui ne tourne pas ; Explain sur un graphe de connaissance ; la section des règles du fichier, que l'analyseur écarte

La limite de ce chapitre est celle que ma directive, à la section 4, porte déjà : *entités, relations, une ontologie, règles*. Le fichier à ce commit n'a pas de règles. Voici le fichier avec une section de règles. Qu'en fait l'analyseur ?

HARO

```
cRules = 'knowledge "pairs"' + nl + nl + "facts" + nl + "    margherita | pairs-with | chianti" +
nl + nl +
         "rules" + nl + "    pairs-with | symmetric" + nl
oRul = new stzKnowledgeGraph("pairs")
aM = oRul.ImportKnow(cRules)
? aM[:merged]
```

```
#--> 1
? len(oRul.Ontology())
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Le fichier fusionne, `1`, et l'ontologie est vide, `0`. L'analyseur reconnaît l'en-tête `rules`, et n'a de gestionnaire pour aucune des lignes qui le suivent. Elles sont écartées : l'ontologie en tient `0`, et aucun refus ne revient. La même loi sous l'en-tête `ontology` donne une ontologie de longueur 1. Les règles vivent ailleurs, comme données dans une règle attachée à un graphe, ce qui est le sujet des chapitres 29 et 30, et dans un fichier séparé qui nomme des fonctions de règle. Dans ce fichier il y a des faits et des lois. J'imprime « faits et lois », et j'envoie les règles à ces deux chapitres. Le fichier unique qui contient des règles, que promet la conception, n'est pas là, et cette section imprime ce manque comme la limite du chapitre.

La loi n'est pas non plus dérivée sur un graphe d'instance. Prenons la même loi symétrique et le même fait que sur le monde par défaut. L'inverse y est-il ?

HARO

```
StzKnowledgeGraphQ("pairs") {
        ConstrainRelation("pairs-with", :Symmetric)
        KnowRelation("margherita", "pairs-with", "chianti")
        ? RelationHasLaw("pairs-with", "symmetric")
        #--> 1
        ? Query([ "chianti", "pairs-with", "margherita" ])
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

La loi est enregistrée, `1`, et l'arête inverse n'y est pas, `0` ! L'en-tête de la classe le dit : `only :unique and :transitive are acted on here` ; la symétrie est seulement enregistrée. Quatre autres limites, chacune avec son exécution.

Une seule relation entre deux choses. Le graphe stocke une arête par paire de nœuds. Admet-il une seconde relation entre les deux mêmes entités, `amina` et `bread` ? Exécutez-le :

HARO

```
oShop = new stzKnowledgeGraph("shop")
oShop.KnowRelation("amina", "sells", "bread")
aV = oShop.Admit("amina", "buys", "bread", [ :source = "till" ])
? aV[:admitted]
#--> 0
? aV[:why]
#--> Can't add fact: 'amina' and 'bread' already carry the edge 'sells' (stzGraph stores one edge
per node pair).
```

✓ exécuté : a affiché ce que ce bloc promet

La seconde relation est refusée, `0`, avec la phrase propre de la bibliothèque, `Can't add fact: 'amina' and 'bread' already carry the edge 'sells' (stzGraph stores one edge per node pair)`. Un modèle de connaissance qui ne peut pas tenir *vend* et *achète* entre les deux mêmes entités est une vraie limite. Le code le dit : `known limitation, revisit with multi-edge support`.

Une question à deux étapes ne répond qu'à sa première étape. `QueryPath` suit-il son second motif, et le prédicat d'une requête peut-il être une variable ? Exécutez-le :

HARO

```
StzKnowledgeGraphQ("coop2") {
        KnowRelation("amina", "sells", "couscous")
        KnowRelation("couscous", "made-from", "flour")
        ? @@( QueryPath([ [ "amina", "sells", "?p" ], [ "?p", "made-from", "?m" ] ]) )
        #--> [ "couscous" ]
        ? @@( Query([ "amina", "?p", "?o" ]) )
        #--> [ ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le chemin répond `[ "couscous" ]`, la première étape et rien de plus : `QueryPath` ignore le reste de ses motifs, donc ce n'est pas encore une requête à plusieurs étapes. Le prédicat variable répond `[ ]`.

Une ligne qui n'est pas un fait. Quatre lignes, dont trois ne sont pas des faits : deux champs, quatre champs, et aucune barre. Que refuse l'analyseur ?

HARO

```
cBad = 'knowledge "t"' + nl + nl + "facts" + nl + "    a | likes | b" + nl + "    c | likes" + nl
+
       "    d | likes | e | f" + nl + "    g likes h" + nl
oT = new stzKnowledgeGraph("t")
aBad = oT.ImportKnow(cBad)
? aBad[:merged]
#--> 1
? @@(aBad[:refused])
#--> [ ]
```

✓ exécuté : a affiché ce que ce bloc promet

Rien. Le fichier fusionne, `1`, et la liste des refus est `[ ]` ; l'analyseur se tait et les trois lignes disparaissent ! Le document de conception promet que le fichier est « validé au chargement ». À ce commit l'analyseur ne garde qu'une ligne ayant exactement trois champs.

Le dialogue fondateur de 2022. La narration raconte la programmation de la connaissance par un dialogue : une pomme comme fruit et comme entreprise, un propriétaire, des relations demandées comme des questions. À ce commit sa première ligne ne tourne pas. Le texte, que je montre et ne lance pas, est `@Q("Apple").IsA(:Fruit)`. Le 8 octobre 2026 il s'est arrêté sur `Calling Function without definition: @isfruit` ; je n'ai pas gardé cette exécution comme bloc. La classe de chaîne de caractères lit le prédicat comme du langage naturel et cherche une fonction de ce nom. Deux autres formes s'arrêtent ailleurs. Les lignes de promesse de la narration sont périmées. La forme la plus proche qui tourne garde le sens. Le monde connaît-il un nom à deux significations ?

HARO

```
StzKnow(:Apple, :Company)
StzKnow(:Apple, :Fruit)
? @@( WhatIs(:Apple) )
#--> [ "company", "fruit" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Il connaît les deux : `[ "company", "fruit" ]`.

L'étude du dossier dit déjà que le dialogue a été gardé en marge, et que le centre s'est déplacé vers le fichier. Le chapitre garde le fichier comme ouverture, et le dialogue de 2022 reste en marge.

Ce qui est conçu : des règles dans le fichier ; la dérivation sur n'importe quel graphe ; une requête à plusieurs étapes ; plusieurs relations entre deux choses ; l'héritage de classes. `DefineClass` écrit un fait, et aucun héritage n'en découle. Le plus proche qui a tourné, dans un essai antérieur dont je n'ai pas gardé l'exécution, est de déclarer `is-a` transitif et d'énoncer la classe comme un fait de cette relation. Et l'hygiène de la connaissance de la conception au-delà de la provenance, les requêtes bornées, les graphes à portée et la révision avec retour arrière, n'a aucune méthode de ce nom.

Ce qui est absent ou cassé : `Explain()` sur un graphe de connaissance, qui appelle `ApplyInference`, une fonction définie nulle part. Et `ValidateOntology()`, qui renvoie 1 quel que soit le contenu de l'ontologie, comme son propre commentaire le dit, `known defect`.

Une note sur l'histoire du document de conception. Son amendement du 22 août 2026 dit qu'aucun fichier de connaissance n'existe nulle part dans l'arbre, et que c'est de l'hygiène et non un manque. À ce commit ce n'est plus vrai : il y en a quatre-vingt-dix-neuf, tous dans le module d'éducation.

Voilà la limite, imprimée. Voyons maintenant les six portes.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Une classe et un fichier texte : `Know`, `KnowRelation`, `ConstrainRelation`, `Prove`, `Query`, `WriteToKnowFile`, `ImportKnow`. Le fichier est comparable et se livre avec le programme.
L'analyste	La page de l'offre le nomme : ce que vous déclarez, ce sont les entités, les règles et les flux de votre organisation, dans un fichier de connaissance en texte brut. Lisez « règles » comme la limite ci-dessus. Pour commencer, dans le cours : le chapitre intitulé « Enseigner un monde ».
Le concepteur	Le dossier est silencieux. Ni la page de l'offre ni l'étude ne relient le fichier de connaissance à la porte du concepteur, et je n'en invente pas.
Le décideur	Le fichier est du texte brut qu'un responsable peut lire. Chaque fait peut porter une source et une confiance. Une affirmation refusée est gardée, nommée et jamais résolue. Une réponse vient avec sa preuve, et un non est certain. La deuxième loi de la conception dit qu'aucune capacité ne doit exiger rien de payant ni de distant. Je n'ai pas exécuté le fichier hors ligne.
Le formateur	Tout le programme d'éducation est fait de fichiers de connaissance. Un apprenant enseigne un monde, puis l'interroge, et la garde du projet vérifie le fichier du monde, le fichier de l'agent et une réponse imprimée. Une institution s'adapte par surcouche, jamais par fourche.
L'agent	Votre mémoire est un graphe de connaissance avec provenance, et votre monde est un fichier que vous chargez et interrogez.

9. EXERCICE

Voici un monde avec deux faits, et une question à laquelle il ne peut pas encore répondre. Votre promesse est la ligne qui prouve que la classe 3b fait partie du primaire, avec la chaîne écrite en entier. Changez le monde, et non la question.

HARO

```
cWorld = 'knowledge "school"' + nl + nl + "facts" + nl +
         "    class-3b | part-of | grade-3" + nl +
         "    grade-3 | part-of | primary" + nl
oW = new stzKnowledgeGraph("school")
oW.ImportKnow(cWorld)
```

```
? oW.Prove([ "class-3b", "part-of", "primary" ])[:narration]
# this prints a refusal. Your promise: the line proving that class-3b is part of primary,
# with the chain written out. Change the world, not the question.
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le et voyez ce qui s'imprime. Si vous changez la question pour demander au sujet du grade 3, il prouvera quelque chose, et ce ne sera pas la ligne que vous avez promise. Puis posez la question inverse, si le primaire fait partie de la classe 3b, une fois que votre monde a sa loi, et voyez si la loi ajoute un fait dans le mauvais sens. Je ne vous dis pas la ligne à ajouter.

Je vous laisse celui-ci. La section 7 lance un fichier de quatre lignes, dont trois ne sont pas des faits, et imprime ce que l'analyseur en dit. Voici ce fichier de nouveau, avec un seul changement : il interroge aussi le graphe sur trois des lignes, et je n'imprime aucune des réponses.

HARO

```
cBad = 'knowledge "t"' + nl + nl + "facts" + nl + "    a | likes | b" + nl + "    c | likes" + nl
+
       "    d | likes | e | f" + nl + "    g likes h" + nl
oT = new stzKnowledgeGraph("t")
aBad = oT.ImportKnow(cBad)
? aBad[:merged]
? @@(aBad[:refused])
? oT.Query([ "a", "likes", "b" ])
? oT.Query([ "d", "likes", "e" ])
? oT.Query([ "g", "likes", "h" ])
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez-le. Lisez `merged`, lisez `refused`, lisez les trois requêtes, et trouvez quelles lignes le graphe détient et ce que l'analyseur vous a dit des autres. Posez la promesse du document de conception, un fichier « validé au chargement », section 4, à côté de ce que vous trouvez ; la section 7 dit ce que l'analyseur garde. Je ne vous donne pas la réponse, et je n'ai pas corrigé l'analyseur. Quand vous l'aurez, vous saurez pourquoi un fichier qui fusionne n'est pas un fichier qui a été validé.

Le chapitre suivant porte sur les langages de motifs : un motif est une règle sur la forme des données, écrite comme des données, et la même habitude sert tous les genres de données. Ici, une loi sur une relation était une ligne dans un fichier ; là, la règle porte sur la forme des données elles-mêmes.

OÙ SONT LES PREUVES

NARRATION	`stz-knowledge-programming-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
NARRATION	`stz-rules-as-graph-rules-narration.md`
SUR LE SITE	offering
DÉCISIONS DE L'ÉTUDE	12.2, 12.6, 12.12, 12.13, 12.19

PARTIE III · CHAPITRE 15

Les langages de motifs

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 13 bloc(s) sur 13 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je pars de l'expression régulière, ce qu'un motif est pour un programmeur qui travaille, et je porte son habitude aux données qui ne sont pas du texte : listes, nombres, tableaux, matrices, temps et graphes. Je fais le même petit geste dans chaque support, un constructeur, un motif dans une chaîne de caractères, `Match`, et je vous montre tout de suite que la grammaire n'est pas tout à fait une. Je dis ce que les gardes prouvent et là où la famille est encore en développement, et je garde sa seule tension pour la limite : un mot que le langage ne connaît pas n'est pas encore refusé. Le principe est celui-ci : un motif est une règle sur la forme des données, écrite comme des données, et la même habitude sert tous les genres de données.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Pour un programmeur qui travaille, un motif est une expression régulière : une notation compacte de la forme d'une chaîne de caractères, remise à un moteur qui dit si le texte lui correspond, et où. Les moteurs sont anciens et bien connus. Celui qui est sous cette bibliothèque est PCRE2, et le site le mesure contre deux autres.

Pour les données qui ne sont pas du texte, le terrain emploie d'autres outils, un par support et aucun partagé. La mise en correspondance structurelle de motifs dans les langages de la famille ML, en Rust, et depuis Python 3.10 en Python. Les vérifications de schéma pour les listes et les enregistrements. Les bibliothèques de qualité des données pour les tableaux. La reconnaissance de motifs de lignes en SQL,

et les moteurs de flux, pour les séquences dans le temps. Et les langages de requête de graphes, dont le cœur est un motif de chemin. C'est une connaissance générale, bien établie ; le dossier ne contient aucune étude datée d'aucun d'eux, et je n'en invente aucune.

L'étude de terrain du domaine de projets lui-même, d'août 2026, donne le profil des petits langages qui ont survécu. Ce profil est une grammaire fermée, une question de domaine, du texte brut, une surface minuscule, une spécification avec une suite de conformité, des décennies de stabilité. Un langage de motifs est exactement cela. Une question, ces données correspondent-elles, en quelques caractères. Ce que le terrain n'offre pas, c'est la même notation d'un support à l'autre.

Vient ensuite le premier principe.

2. LE PREMIER PRINCIPE

Un motif est une règle sur la forme des données, écrite comme des données. La même habitude sert tous les genres de données.

La phrase à laquelle reviennent les articles de la bibliothèque, dans chaque support, est celle de l'article sur les nombres : au lieu d'indiquer à la machine comment tester une condition, vous énoncez ce que vous voulez dire. Mon étude du dossier, qui n'est pas l'étude de terrain ci-dessus, porte aussi une phrase de moi, « pour la première fois dans l'histoire de l'informatique ». Je ne l'ai trouvée dans aucun texte de la bibliothèque, donc le site ne la revendique pas. Je la garde ici comme ce que j'ai dit, et je n'affirme que ce qui est mesuré : une seule grammaire en forme de regex pour sept genres de données dans une seule bibliothèque, le texte et six qui ne sont pas du texte, avec la même habitude de petite fonction. Et je vais vous montrer tout de suite que la grammaire n'est pas tout à fait une.

3. LA FORME SOFTANZA

Un motif appelé par son nom est un catalogue fermé. La bibliothèque a un fichier de données de motifs nommés. Que dit-elle quand vous demandez un nom qu'elle ne tient pas ? Exécutez-le :

```
HARO
```

```
? rx(pat(:isoDate)).Match("2023-12-25")
#--> 1
? rx(pat(:isoDate)).Match("25/12/2023")
#--> 0
? rx(pat(:isoDate)).Explain()
#--> Matches ISO 8601 date format (YYYY-MM-DD)
try
        pat(:banana)
catch
        ? cCatchError
done
#--> The pattern name you provided does not exist in stzRegexData file.
```

✓ exécuté : a affiché ce que ce bloc promet

Elle refuse, et dit pourquoi : `The pattern name you provided does not exist in stzRegexData file.`

Un motif peut aussi répondre à la valeur à moitié écrite, ce que veut un formulaire pendant que la personne tape :

HARO

```
StzRegexQ("^\d{3}-\d{2}-\d{4}$") {
        ? MatchAsYouType("123-45")
        #--> 1
        ? MatchAsYouType("abc")
        #--> 0
        ? IsPartialMatch("123-45")
        #--> 1
        ? IsPartialMatch("123-45-6789")
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

Voici maintenant le geste qui donne son titre au chapitre. Prenons une liste. Son motif s'écrit entre crochets, et les comptes, l'imbrication et la négation sont écrits, non bouclés. `@N` représente un nombre, `@S` une chaîne de caractères, `+` un ou plus, `@N1-3` un à trois, et `@!` le non. `[@N1-3, @S]` accepte-t-il quatre nombres avant la chaîne de caractères ? Le deuxième appel répond :

HARO

```
? Lx("[@N+, @S]").Match([1, 2, 3, "end"])
#--> 1
? Lx("[@N1-3, @S]").Match([1, 2, 3, 4, "end"])
#--> 0
? Lx("[@S, [@N+]]").Match(["tea", [1, 2, 3]])
#--> 1
? Lx("[@S, [@N+]]").Match(["tea", ["x"]])
#--> 0
? Lx("[@!N, @N]").Match(["a", 1])
#--> 1
? Lx("[@!N, @N]").Match([1, 1])
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Non : le deuxième appel imprime 0, car quatre nombres sont un de trop. Chaque appel ici utilise un objet neuf, pour une raison que donne la section 7. Le chapitre 11 a montré quatre appels du même langage, et je ne les répète pas.

Les nombres ont des propriétés, et une propriété est du sens, non de la procédure. Quels nombres entre 1 et 500 sont parfaits ? La dernière ligne fait passer un motif d'un test à une recherche :

HARO

```
? Nx("{@Property(Even) & @Property(Prime)}").Match(2)
#--> 1
? Nx("{@Property(Even) & @Property(Prime)}").Match(4)
#--> 0
? Nx("{@Digit(:unique)+}").Match(1234)
#--> 1
? Nx("{@Digit(:unique)+}").Match(1223)
#--> 0
? Nx("{@Relation(Mod:5=0)}").Match(10)
#--> 1
? @@( Nx("{@Property(Perfect)}").MatchingNumbersBetween(1, 500) )
#--> [ 6, 28, 496 ]
```

✓ exécuté : a affiché ce que ce bloc promet

Trois nombres répondent, et aucune boucle n'a été écrite.

Les tableaux : une règle de qualité des données est une phrase du langage. La personne qui possède les données peut la lire. Niamey apparaît deux fois, donc les villes ne sont pas uniques, et la négation de cela est vraie :

HARO

```
oT = new stzTable([
        [ :ID, :NAME, :AGE, :CITY ],
        [ 1, "Ali", 28, "Niamey" ],
        [ 2, "Sara", 32, "Paris" ],
        [ 3, "Omar", 41, "Niamey" ],
        [ 4, "Awa", 25, "Dakar" ]
])
? Tbx("{cols(4) & unique(id) & avgcol(age:>30)}").Match(oT)
#--> 1
? Tbx("{cols(4) & unique(city)}").Match(oT)
#--> 0
? Tbx("{@!unique(city)}").Match(oT)
#--> 1
? Tbx("{sorted(id) & numeric(age) & alphabetic(name)}").Match(oT)
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Un motif est une donnée, donc les motifs se composent : on en construit un à partir de motifs.

HARO

```
oA = new stzTablex("{cols(4)}")
oB = new stzTablex("{rows(4)}")
oAnd = oA.And_(oB)
? oAnd.Pattern()
#--> {cols(4) & rows(4)}
? oAnd.Match(oT)
#--> 1
? oA.Not_().Pattern()
#--> {@!cols(4)}
? oA.Not_().Match(oT)
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Une matrice : un concept, carrée et symétrique, tient en une ligne.

HARO

```
? Mx("{shape(square) & property(symmetric)}").Match([ [1,2,3], [2,4,5], [3,5,6] ])
#--> 1
? Mx("{shape(square) & property(symmetric)}").Match([ [1,2,3], [4,5,6], [7,8,9] ])
#--> 0
? Mx("{size(2x3)}").Match([ [1,2,3], [4,5,6] ])
#--> 1
? Mx("{property(identity)}").Match([ [1,0], [0,1] ])
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Le temps, dans la part qui s'exécute, écrit ici avec la classe plutôt qu'avec sa petite fonction `Tmx`. Une durée doit satisfaire une règle : d'une à deux heures, sur un pas d'un quart d'heure. Laquelle de trois durées convient : quatre-vingt-dix minutes, quatre-vingts minutes ou trois heures ?

HARO

```
oTm = new stzTimex("{@Duration(1h..2h:15min)}")
oTm {
        ? Match(new stzDuration("1 hour 30 minutes"))
        #--> 1
        ? Match(new stzDuration("1 hour 20 minutes"))
        #--> 0
        ? Match(new stzDuration("3 hours"))
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

Seules quatre-vingt-dix minutes conviennent : quatre-vingts sont dans l'intervalle et hors du pas, et trois heures sont dehors.

Un graphe : un motif se lit comme le chemin qu'il cherche. Les étiquettes correspondent sans égard à la casse. Le sens compte-t-il ? La deuxième ligne répond :

HARO

```
g = new stzGraph("Order")
g {
```

```
        AddNodeXT(:a, "Placed")
        AddNodeXT(:b, "Cooking")
        AddNodeXT(:c, "Served")
        AddEdgeXT(:a, :b, "starts")
        AddEdgeXT(:b, :c, "finishes")
}
? (new stzGraphex("{@Node(placed) -> @Edge(starts) -> @Node(cooking)}", g)).Match(g)
#--> 1
? (new stzGraphex("{@Node(served) -> @Node(placed)}", g)).Match(g)
#--> 0
? (new stzGraphex("{@Node(placed) -> @!Node(refunded)}", g)).Match(g)
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Oui : le chemin de `served` à `placed` ne trouve rien, et la ligne imprime 0.

Enfin, l'habitude. Le motif est une donnée dans chaque langage, et un langage qui expose ses jetons peut être relu par une personne et par un agent d'IA. Cinq des six qui ne sont pas du texte le font ; en voici trois :

HARO

```
? len( Lx("[@N, @S]").Tokens() )
#--> 2
? Lx("[@N, @S]").Tokens()[2][:type]
#--> string
? Nx("{@Property(Even) & @Property(Prime)}").Tokens()[1][:type]
#--> conjunction
? len( Mx("{shape(square) & property(symmetric)}").Tokens()[1][:conditions] )
#--> 2
```

✓ exécuté : a affiché ce que ce bloc promet

Les graphes n'ont ni `Tokens`, ni `Pattern`, ni `Explain` qui fonctionne à ce commit, et pas de petite fonction non plus. Il n'y a pas de `Gx`, et il n'y a pas de `Tx` pour le temps, bien que le tableau de l'étude nomme les deux. Les petites fonctions sont `rx`, `Lx`, `Nx`, `Tbx`, `Mx` et `Tmx`.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé qu'un motif énonce un sens et non une procédure. Cette décision est dans les articles, et les articles sont les miens. Les mêmes ont décidé de porter l'idée de la regex aux données qui ne sont pas du texte avec une seule habitude : des accolades, un `@` pour un jeton, `&`, `|`, `@!` pour le non, `@cs:` pour la casse. Les listes gardent les crochets. Les cinq autres utilisent des accolades. C'est la première correction honnête de l'expression « une seule grammaire ».

J'ai refusé de réinventer la notation pour le texte. Les correspondances simples sont gardées : une chaîne de motif classique, les groupes nommés et les captures fonctionnent comme écrits. Des noms plus clairs sont posés autour. Le travail plus ancien sur le nommage est gardé comme histoire, pour sa valeur éducative. Je tiens cela pour la bonne façon de mettre à la retraite un mécanisme.

J'ai refusé, pour l'instant, de construire les classes sœurs. Les articles prévoyaient des classes dans lesquelles les motifs sont génétiques, linguistiques ou quantiques. L'article dit qu'elles sont dans un prototype précoce et non dans la bibliothèque, et une recherche dans le dossier de la regex des mots « induce », « from example » et « infer » ne trouve rien. L'étude date la découverte de leur but de deux ans plus tard, dans le protocole de réponse (la façon dont une personne peut répondre par des exemples) : l'induction à partir d'exemples, et les candidats superposés. C'est le terrain du chapitre 18, et le refus du chapitre 9.

J'ai refusé le constructeur déclaratif comme titre phare. Les constructeurs sont les classes qui écrivent une regex à partir d'étapes nommées dans le code, et non d'une chaîne de caractères. Dix-huit des cinquante et un fichiers du dossier de tests du constructeur sont marqués retirés : l'analyseur d'un motif déclaratif est en attente. L'exemple d'ouverture de l'article, `CanContainAChar(:Between = ...)`, lève une erreur. Ce qui tourne, c'est le constructeur conditionnel, et l'objet de look-around. Je ne bâtis pas le chapitre sur la forme phare du constructeur.

Je n'ai pas encore refusé un mot que le langage ne connaît pas. C'est la tension de ce chapitre, et je la garde pour la section 7.

5. LE TRIBUNAL

`00_regex_narrated` prouve les ancres et la recherche, les classes, les quantificateurs exacts, les groupes, les quatre types de correspondance, la correspondance partielle pendant la frappe, et les positions comme données. `00_numbrex_narrated` prouve premier, parfait, Fibonacci, pair et impair, carré et composé, chacun avec un nombre qui correspond et un qui ne correspond pas. Son en-tête consigne une régression où `Match()` renvoyait autrefois TRUE pour toute entrée. `tablex_narrated` prouve l'analyseur, le comptage, l'unicité, le tri, les types, les sommes et les moyennes, les trois connecteurs, les séquences, et la construction de motifs à partir de motifs. Son en-tête nomme la faute qu'il a tuée. L'analyseur découpait avec une position de fin à travers une fonction qui compte, et environ quatre-vingt-dix motifs étaient analysés en un jeton du bon type sans valeur. `00_matrex_narrated` prouve la taille exacte, toute taille et la forme carrée. Son en-tête consigne que `Match()` renvoyait autrefois FALSE pour toute entrée. `00_graphex_narrated` prouve qu'un motif se compile en un graphe, les étiquettes présentes et absentes, une séquence comme sous-suite, les règles de casse, la négation, l'alternance, les propriétés numériques, et le cache par signature de graphe.

Notez l'emplacement : la garde des tableaux vit dans le dossier de la regex, et son nom n'a pas de préfixe `00_`.

Il n'y a pas de garde narrée pour les listes, le temps ou les constructeurs de regex. Les tribunaux de ceux-ci sont des fichiers classiques sans promesses utilisables. Les tests des listes sont placés après une définition de fonction, de sorte que le langage hôte les lit comme le corps de la fonction et n'en exécute aucun. J'ai lancé une copie avec la fonction déplacée à la fin, et je n'ai pas gardé cette exécution comme bloc. Soixante vérifications se sont exécutées, et deux d'entre elles contredisent l'attente

du fichier lui-même. Le fichier classique du temps dit qu'une promesse est TRUE et l'exécution imprime 0, et le dossier central l'a consigné le 1er octobre 2026 comme encore ouvert.

Vient ensuite le terrain, et où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`00_regex_narrated` 28 assertions
base/test/regex/00_regex_narrated.ring

`00_numbrex_narrated` 15 assertions
base/test/numbrex/00_numbrex_narrated.ring

`tablex_narrated` 95 assertions base/test/regex/tablex_narrated.ring

`00_matrex_narrated` 8 assertions
base/test/matrex/00_matrex_narrated.ring

`00_graphex_narrated` 27 assertions
base/test/graphex/00_graphex_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Chaînes de caractères. Le terrain mène en sûreté et en vitesse, et Softanza mène en nommage. La page du site pour ce domaine le dit elle-même : elle est derrière les moteurs modernes en sûreté et en débit, sans JIT et sans moteur de correspondance en temps linéaire. Le contrôle contre le retour arrière catastrophique n'est pas atteignable depuis Haro, bien que le code pour les deux soit déjà compilé dans le binaire. L'en-tête du moteur confirme que le durcissement existe en dessous : une limite de correspondances, une limite de récursion, des plafonds de longueur.

Listes, nombres, temps, graphes. Le terrain a des outils uniques plus anciens et mieux établis. L'étude le dit sans les classer : un outil a des motifs de listes, l'unicité et des intervalles à pas, et trois langages ont la mise en correspondance structurelle. Les langages de requête de graphes existent, et le traitement d'événements existe pour le temps. Ce que le terrain n'a pas, et ce que le dossier revendique, c'est l'habitude unique d'un support à l'autre.

Tableaux. Les bibliothèques de qualité des données mènent en portée et en écosystème. Ici une règle est une phrase que la personne qui possède les données peut lire. La perte est qu'elle ne s'exécute que sur `stzTable`.

Là où la famille est encore en développement. Les en-têtes de classe impriment leurs limites connues comme des avertissements : pour les matrices et les graphes. C'est rare, et je le dis. Graphes : les quantificateurs sont analysés mais pas encore utilisés par la correspondance. Matrices : des manques connus aujourd'hui, chacun porté comme un avertissement sur sa méthode.

Une date pour le bogue récurrent de la famille. La même cause racine a frappé la famille langage par langage : les analyseurs découpaient avec une position de fin à travers une fonction qui compte. Les nombres et les matrices ont été réparés le 14 juin 2026, les tableaux le 5 octobre, et les graphes dans une correction dont le fichier ne donne pas la date. Le temps le porte encore. C'est la preuve que la famille partage une seule conception et les habitudes d'un seul analyseur ; c'est aussi une raison probable pour laquelle le temps ne correspond pas encore.

Vient ensuite la limite.

7. LA LIMITE

CONSTRUIT	des motifs pour le texte, les nombres et les tableaux, chacun avec une garde ; la même petite habitude d'un support à l'autre (un constructeur, un motif dans une chaîne de caractères, Match, Tokens)
CONÇU	un seul moteur de motifs partagé par tous les supports ; le motif comme côté condition de la cascade ; le motif écrit à partir d'exemples
ABSENT	le refus d'un mot que le langage de motifs ne connaît pas, dans les six supports qui ne sont pas du texte ; des séquences sur une ligne de temps,

> dans le temps ; une narration et une petite fonction pour les graphes ; les classes sœurs génétique, linguistique et quantique

Ce qui est construit : les chaînes de caractères, les nombres et les tableaux, chacun avec une garde et une petite fonction. L'habitude partagée.

Ce qui est construit en partie : les listes, les matrices, le temps et les graphes, dans cet ordre.

- **Listes** : les types, les comptes, l'imbrication, la négation, l'alternance et les ensembles s'exécutent. Les intervalles à pas lèvent une erreur. Dans une exécution que je n'ai pas gardée comme bloc, le drapeau d'unicité a été ignoré, de sorte que `[@N2-3{1; 2; 3}U]` a accepté `[1, 1]`, bien que la classe ait du code pour une vérification d'unicité. Le joker de l'article, `@A`, n'est pas un jeton ; le vrai joker est `@$`. Et un objet réutilisé répond depuis un cache dont la clé est la longueur de la liste et la somme des codes de ses caractères, ce qui ne dépend pas de l'ordre. Un objet neuf répond 0 pour `["hello", 1]`, et le même objet après `[1, "hello"]` répond 1.
- **Matrices** : la taille, la forme, l'élément et trois propriétés s'exécutent. L'en-tête de la classe liste les termes qui acceptent toute matrice : `row`, `col`, `pattern`, `determinant` et `sum`. Une matrice de deux par deux est acceptée par `{row(99)}`.
- **Temps** : les instants et les durées avec un pas s'exécutent. Dans une exécution que je n'ai pas gardée comme bloc, un intervalle sans pas a levé une erreur. Et les séquences sur une ligne de temps ne s'exécutent pas du tout : voici une ligne de temps avec une réunion, une pause et un déjeuner, et le motif que l'article promet. Correspond-il ?

HARO

```
oTl = new stzTimeLine("2025-10-22", "2025-10-22")
oTl {
        AddInstant("Meeting", "2025-10-22 09:00:00")
        AddSpan("Break", "2025-10-22 10:00:00", "2025-10-22 10:15:00")
        AddInstant("Lunch", "2025-10-22 12:00:00")
}
o1 = new stzTimex("{@Event(Meeting) -> @Duration* -> @Event(Break)}")
? o1.MatchPartial(oTl)
#--> 0
o2 = new stzTimex("{@Event(Lunch)}")
```

```
? o2.MatchPartial(oTl)
#--> 0
? @@( o2.Tokens()[1][:label] )
#--> "Lunch)"
```

✓ exécuté : a affiché ce que ce bloc promet

L'article promet que le premier est TRUE. Le deuxième montre que même un événement unique que la ligne de temps contient ne correspond pas ! Le troisième montre pourquoi : l'étiquette du jeton est `"Lunch)"`, lue avec sa parenthèse fermante. C'est le bogue partagé ; le réparer dans le temps est l'étape suivante.

- **Graphes** : les étiquettes dans l'ordre le long d'un chemin, la négation, l'alternance et les propriétés numériques s'exécutent. La flèche signifie *plus tard sur un chemin*, et non *suivant*. Les quantificateurs (`+`, `*`, `?` ou un nombre après un jeton) sont lus et ignorés. Le graphe de la section 3 a trois nœuds ; ici `@Node10`, un nœud avec le quantificateur dix, est demandé entre eux. Le motif correspond-il ?

HARO

```
? (new stzGraphex("{@Node(placed) -> @Node(served)}", g)).Match(g)
#--> 1
? (new stzGraphex("{@Node(placed) -> @Node10 -> @Node(served)}", g)).Match(g)
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Cooking se trouve entre le premier et le dernier, donc la première réponse est le sens de la flèche. La seconde répond 1 aussi ! Le quantificateur est ignoré. Le cache de cette classe est indexé par les comptes de nœuds et d'arêtes et les identifiants de nœuds. Un second graphe avec les mêmes identifiants et d'autres étiquettes est servi depuis le cache, jusqu'à `ClearCache()` : la réponse peut être périmée.

Le refus que la famille n'a pas encore fait. La loi du chapitre 11 est qu'un langage est déclaré et fermé, avec des refus. Que répond la famille quand on lui demande un mot qu'elle ne connaît pas ? Exécutez-le :

HARO

```
? Nx("{@Banana(3)}").Match(7)
#--> 1
? Nx("{@Banana(3)}").Match(1000003)
#--> 1
? @@( Nx("{@Banana(3)}").Tokens() )
#--> [ ]
? Nx("{@Property(Banana)}").Match(7)
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Un motif avec un mot que le langage ne connaît pas correspond à tous les nombres ! Les jetons sont [ ], et rien n'est levé. Les langages des matrices et des tableaux renvoient un jeton ERROR que rien ne lève, et les langages des nombres et du temps renvoient une liste vide. La session de la charte de Haro l'a consigné comme constat, que le dossier central a relayé le 28 septembre 2026 : les six classes de motifs acceptent en silence les jetons inconnus. Je l'ai reproduit à ce commit, et la demande n'est pas close. Un langage fermé qui ne peut pas refuser n'est pas encore la loi du chapitre 11. Le chapitre l'imprime comme un constat du tribunal, et non comme une dette de moi. Je le tiens pour mon point de vue. L'exécution ci-dessus, à ce commit, est ce qui le soutient.

Ce qui est conçu : un seul moteur de motifs partagé par tous les supports. La conception du moteur dit que le moteur à retour arrière, le solveur de contraintes et le compilateur de motifs sont les mêmes dans tous les domaines, et que seul le vocabulaire change. Son étape est planifiée. À ce commit le vocabulaire change et le code aussi. Le fichier propre du moteur nommé *pattern* est autre chose : les palindromes, les répétitions et les suites arithmétiques et géométriques. Le motif comme côté condition de la cascade, plusieurs facultés qui se déclenchent l'une après l'autre sur un état partagé, est le troisième échelon du chapitre 9, et conçu.

Ce qui est absent : l'induction à partir d'exemples, qui est réservée. Les classes sœurs génétique, linguistique et quantique, qui sont de la vision seulement. L'explication d'un motif sans nom, puisque `rx("[a-z]+").Explain()` lève une erreur parce que la classe d'analyse n'a jamais été construite. Et tout usage des classes de motifs par le reste de la bibliothèque : en dehors du dossier de la regex, des tests et de la documentation, aucune classe ne prend un motif. `stzList`, `stzNumber`, `stzTable`,

`stzMatrix` et `stzGraph` n'ont pas de `Matches(pattern)`. Dans la documentation, le cours d'éducation les utilise, dans un chapitre et un exercice dont la réponse est vérifiée en exécutant le programme de l'apprenant, et la galerie a quelques scripts qui les utilisent.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Une habitude pour sept genres de données, les graphes n'ayant pas encore la petite fonction : un petit constructeur, un motif dans une chaîne de caractères, `Match`. Le motif est une donnée que vous pouvez lire et composer. Servez-vous de ceux qui ont une garde : le texte, les nombres, les tableaux, les matrices et les graphes, ces deux derniers avec les manques de la section 7, et ne bâtissez rien sur le cache d'un motif de liste réutilisé.
L'analyste	Une règle de qualité des données est une phrase du langage des tableaux. La personne qui possède les données peut la lire.
Le concepteur	Un motif énonce une forme que l'interface peut montrer avant que les données n'arrivent, et une valeur à moitié tapée reçoit une réponse en chemin.
Le décideur	Une règle est une ligne et non une procédure. Elle peut être relue, et son verdict est 1 ou 0. Dites à côté qu'un motif avec un mot inconnu n'est pas refusé, et qu'une flèche de graphe signifie *plus tard*.
Le formateur	Le cours enseigne les motifs en deux chapitres, en anglais, en français, en arabe et en haoussa. L'échelle est un programme : texte, liste, nombre.
L'agent	Un motif est une petite notation fermée à laquelle un agent peut être tenu et qu'il peut relire. On vous dit qu'aucun des six supports qui ne sont pas du texte ne peut encore refuser un mot qu'il ne connaît pas, et que l'écriture d'un motif à partir d'exemples est réservée.

9. EXERCICE

Trois petits tableaux sont donnés. Écrivez un seul motif qui n'accepte que le premier. Avant de rien exécuter, écrivez sous chacune des trois lignes imprimées ce que vous attendez et pourquoi.

```
HARO
```

```
t1 = new stzTable([ [ :ID, :EMAIL ], [ 1, "ali@x.org" ], [ 2, "sara@x.org" ] ])
t2 = new stzTable([ [ :ID, :EMAIL ], [ 1, "ali@x.org" ], [ 1, "sara@x.org" ] ])
t3 = new stzTable([ [ :ID, :EMAIL ], [ 1, "ali@x.org" ], [ 2, "" ] ])
# write ONE pattern, then print whether it matches t1, t2, t3, on three lines
cPattern = "{ ... }"
? Tbx(cPattern).Match(t1)
? Tbx(cPattern).Match(t2)
? Tbx(cPattern).Match(t3)
#--> 1
#--> 0
#--> 0
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Une clause rejette le deuxième tableau et une autre rejette le troisième ; laquelle est laquelle ? Et que dirait le motif si l'email était une simple espace ? Je ne réponds pas en premier. Si une promesse se rompt, ne la changez pas : expliquez la sortie.

Je vous laisse celui-ci, exprès, et je le nomme un défaut. Reprenez le bloc des matrices de la section 3 et changez une seule chose dans sa dernière ligne : mettez `{row(99)}` là où se trouve `{property(identity)}`, et gardez la matrice.

HARO

```
? Mx("{row(99)}").Match([ [1,0], [0,1] ])
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Dites ce que vous attendez avant de l'exécuter, puis exécutez-le. Ne changez pas le motif pour que la sortie vous donne raison : la section 7 dit où la bibliothèque imprime ce défaut, et pourquoi. Quand vous l'aurez trouvé, vous saurez quels termes du langage des matrices acceptent toute matrice.

Le chapitre suivant porte sur sept langages et une porte. Une porte est un runtime et jamais une syntaxe, et la porte est une : elle est la réponse à la question « et le langage X ? ».

OÙ SONT LES PREUVES

NARRATION	stznumbrex-pattern-language-for-number.md
NARRATION	stztablex-pattern-langauge-for-tables.md.md
NARRATION	stzlistex-pattern-matching-for-lists.md
NARRATION	stzTimex-A-Pattern-Language-for-Time-in-Softanza.md
NARRATION	stzMatrex-Declarative-Pattern-Matching-for-Matrices.md
SUR LE SITE	languages
DÉCISIONS DE L'ÉTUDE	12.4, 12.18, 12.26

PARTIE III · CHAPITRE 16

Sept langages, une porte

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 10 bloc(s) sur 10 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends les deux leçons du terrain sur l'atteinte d'un autre langage, une évasion ouverte et une frontière sans chemin de sortie, et je place la porte là où elles se rencontrent. J'interroge la porte sur les langages qu'elle connaît, j'y fais passer Python et NodeJS, je lis Python et JavaScript dans un seul graphe de code, et je montre la gouvernance refusant un exécuteur sans posture. Puis je dis ce que j'ai décidé : une seule porte, et pas d'autres intégrations d'hôtes. Et je dis où en est la porte à ce commit : sa gouvernance est conçue, un chemin du retour fermé est absent, et la porte R, en développement, lève une erreur aujourd'hui. Le principe est celui que le chapitre 11 a promis : une facette est une syntaxe d'une grammaire et jamais un runtime, une porte est un runtime et jamais une syntaxe, et la porte est une.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Pour un programmeur qui travaille, atteindre un autre langage depuis celui dans lequel on écrit signifie l'une de deux choses. La première est une interface de fonctions étrangères, un appel dans une bibliothèque partagée. La seconde est un processus enfant, une exécution d'un interpréteur dont on analyse la sortie. Les carnets et les machines virtuelles polyglottes font le même geste avec plus de confort. C'est une connaissance générale ; le dossier ne tient aucune étude datée de ces outils, et le dit lui-même.

L'étude de terrain propre du domaine de projets, d'août 2026, a deux phrases sur ce sujet, et elles tirent en sens opposés. À la sortie, le langage hôte est l'issue de secours, et l'issue de secours est toujours prise. Elle donne deux langages internes morts d'ouverture. À l'entrée, le différenciateur est l'évasion ; un langage que l'on fait doit pouvoir être consommé depuis n'importe quelle pile. Elle donne un langage qui a calé à la frontière de son propre monde.

La leçon du terrain est donc double. Une évasion ouverte détruit un langage fermé. Et un langage sans chemin de sortie meurt à sa propre frontière. Une porte est l'endroit où ces deux leçons se rencontrent.

Il y a une seconde habitude, qui ressemble à la première et n'en est pas une. Les outils qui laissent écrire dans la syntaxe que l'on connaît par-dessus un autre moteur, un transpileur, un second front end pour un seul compilateur, je les appellerai du mot de ce livre, des *facettes*, que le chapitre 12 définit. La note de conception du domaine de projets à leur sujet est franche sur leur coût. La promesse d'une seconde facette est la lisibilité pour un humain qui connaît l'autre langage, et non le transfert de comportement.

2. LE PREMIER PRINCIPE

Une facette est une syntaxe d'une grammaire, et jamais un runtime. Une porte est un runtime, et jamais une syntaxe. La porte est une, et elle est la réponse à « et le langage X ? ».

La phrase propre de la vision, ratifiée en session le 28 août 2026, le dit de la porte : du code étranger dans les langages choisis, gouverné, une porte, sans autres intégrations d'hôtes. L'étude du dossier lit la paire en une ligne que je garde : la facette amène le programmeur, et la porte amène l'écosystème. Le chapitre 11 a promis que ce chapitre dirait la différence ; le chapitre 1, qu'il donnerait les sept couches. Voici la différence.

3. LA FORME SOFTANZA

La liste de la porte est fermée. Les langages étrangers forment un registre, et l'hôte n'y figure pas. Quels langages la porte connaît-elle ? Interrogez-la sur six langages, puis sur l'hôte, sous le nom que les données propres de la bibliothèque lui donnent

encore, celui de l'ancien langage hôte, puis sur un langage qui n'a jamais été sur la liste :

HARO

```
oDoor = new stzExterCode(:python)
aNames = [ "c", "nodejs", "julia", "r", "prolog", "python", "ring", "cobol" ]
for cLang in aNames
        ? cLang + ": " + oDoor.IsLanguageSupported(cLang)
next
#--> c: 1
#--> nodejs: 1
#--> julia: 1
#--> r: 1
#--> prolog: 1
#--> python: 1
#--> ring: 0
#--> cobol: 0
```

✓ exécuté : a affiché ce que ce bloc promet

« Et le langage X ? » a une seule réponse, et pour un langage hors de la liste c'est un refus par son nom :

HARO

```
try
        oNo = new stzExterCode(:cobol)
catch
        ? cCatchError
done
#--> Language 'cobol' is not supported
```

✓ exécuté : a affiché ce que ce bloc promet

La porte l'a refusé par son nom : `Language 'cobol' is not supported !`

Maintenant la porte exécute un vrai runtime. Une note sur la machine d'abord. La porte écrit un script, démarre le runtime avec lui et relit un fichier de résultat. L'emplacement par défaut de la bibliothèque pour Python n'est pas celui où cette machine le garde, aussi la porte reçoit-elle la commande nue `python`, que la machine résout sur son chemin. Node et R se trouvent aux emplacements par défaut de la

bibliothèque. Les blocs qui démarrent un runtime sont marqués, dans le lecteur, comme touchant au système, et ils montrent la sortie qui a été enregistrée à la date de la construction.

Que rapporte un appel à Python ? Le résultat revient sous la structure propre de l'hôte, une liste, et l'appel laisse une trace :

HARO

```
cPython = "python"
oPy = new stzExterCode(:python)
oPy {
        SetRuntimePath(cPython)
        SetCode("
res = { 'numbers': [1, 2, 3, 4, 5], 'mean': sum([1, 2, 3, 4, 5]) / 5 }
        ")
        Execute()
        ? @@( Result() )
        #--> [ [ "numbers", [ 1, 2, 3, 4, 5 ] ], [ "mean", 3 ] ]
        ? CallTrace()[1][:language]
        #--> python
        ? CallTrace()[1][:mode]
        #--> interpreted
        ? isNumber(LastCallDuration())
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le flottant Python `3.0` est revenu sous la forme `3`, dans `[ "mean", 3 ]`. La trace contient aussi une durée. Le bloc imprime seulement `1`, la réponse de `isNumber`, car une durée est une horloge, et une horloge ne se promet pas.

Voilà la raison d'être de tout le volet, en une image. Chaque couche d'une solution est dans le langage qui y est le meilleur, et l'hôte passe les données en texte et les relit en retour. L'article de la bibliothèque énumère les couches : la performance système en C, les systèmes événementiels en NodeJS, le calcul scientifique en Julia, les statistiques en R, le raisonnement logique en Prolog, et l'apprentissage automatique en Python. Il y ajoute le raisonnement d'IA dans un modèle de langage,

et l'intégration de la solution dans l'hôte. Voici deux d'entre elles dans un pipeline, Python calculant les moyennes de deux étudiants et NodeJS décidant. Les seuils sont de moi ; les données sont celles de l'article :

HARO

```
aStudents = [ [ "Alice", [ 85, 90, 78 ] ], [ "Bob", [ 72, 65, 70 ] ] ]
oPy2 = new stzExterCode(:python)
oPy2 {
        SetRuntimePath(cPython)
        SetCode("
students = " + @@(aStudents) + "
res = [ [ s[0], round(sum(s[1]) / len(s[1]), 2) ] for s in students ]
        ")
        Execute()
        aAverages = Result()
}
? @@(aAverages)
#--> [ [ "Alice", 84.33 ], [ "Bob", 69 ] ]
oNode = new stzExterCode(:nodejs)
oNode {
        SetCode("
const avgs = " + @@(aAverages) + ";
const res = avgs.map(a => [ a[0], a[1] >= 80 ? 'Excellent' : 'Needs work' ]);
        ")
        Execute()
        ? @@( Result() )
        #--> [ [ "Alice", "Excellent" ], [ "Bob", "Needs work" ] ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Les données entrent en composant la chaîne de code avec l'afficheur de listes propre à l'hôte, puisque les fonctions de conversion de l'article sont absentes (section 7).

Lesquels, ces sept ? Le titre s'engage sur un nombre, et j'ai tranché le 12 octobre 2026 lequel : les six runtimes étrangers du registre ci-dessus et l'hôte, comme les comptent l'article sur le volet et l'article sur la performance. Le modèle de langage n'en fait pas partie, car il n'est pas du tout une porte de ce volet. Les chapitres 17 et 20 font du modèle un proposeur qui ne peut pas valider, alors que les narrations l'exécutent comme un runtime de première classe par une fonction absente. La section 7 imprime cela comme une limite. Le dossier compte autrement ailleurs :

sept modules de domaine dont un pour un modèle de langage, sept domaines avec l'hôte comme huitième, les six du registre sans l'hôte, les « 7 langages choisis » de la vision qu'elle n'énumère jamais, et les cinq de la conception du moteur avec COBOL. Le registre ci-dessus est celui qui s'exécute.

Une facette n'est pas une porte. Lire un langage étranger n'est pas non plus l'exécuter. Les deux langages que je nomme comme facettes prévues de Haro sont aussi les deux que la bibliothèque sait déjà lire dans un seul graphe de code. Elle ne les exécute pas et ne les écrit pas. Un seul jeu de questions répond sur les deux :

HARO

```
cPy = "class Animal:" + nl +
      "    def speak(self):" + nl +
      "        return '...'" + nl +
      "" + nl +
      "class Dog(Animal):" + nl +
      "    def speak(self):" + nl +
      "        return 'woof'"
cJs = "class Animal { speak() { return '...' } }" + nl +
      "class Dog extends Animal { speak() { return 'woof' } }"
StzPyCodeGraphFromSourceTS(cPy) {
        ? Language()
        #--> python
        ? @@( ParentsOf("Dog") )
        #--> [ "Animal" ]
        ? @@( OwnersOf("speak") )
        #--> [ "Animal", "Dog" ]
}
StzJsCodeGraphFromSourceTS(cJs) {
        ? Language()
        #--> javascript
        ? @@( ParentsOf("Dog") )
        #--> [ "Animal" ]
        ? @@( OwnersOf("speak") )
        #--> [ "Animal", "Dog" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

C'est un analyseur qui produit un index de classes, de méthodes et d'héritage, et non un graphe de programme capable d'imprimer un programme. C'est une coïncidence de l'ordre du travail, et pas encore une facette. Le dossier tient une seule déclaration de moi sur les facettes. Je l'ai faite en session le 8 octobre 2026, et elle dit qu'un lien

doit être fait avec la future vision de Haro et de ses langages à facettes. Dans cette vision, l'ancien langage hôte est déjà là, et Python et JavaScript sont prévus. La charte de Haro sur cette machine parle d'une grammaire et d'un texte. Sa note de conception pour une seconde facette, qui propose et ne tranche rien, recommande des afficheurs lus par des humains avant tout analyseur. Sa propre table énumère les collisions : un lecteur de Python attend une indexation à partir de zéro, et celle de Haro part de un. Le chapitre dit « prévues » seulement avec cette déclaration nommée comme source.

Une porte est gouvernée. La gouvernance a le mot pour le code étranger. `external` est la posture que la bibliothèque définit comme hors processus. Que répond la gouvernance avant que cette posture soit déclarée, et après ? Interrogez-la :

HARO

```
oGov = new stzGovernance("shop")
oGov {
        ? MayExecute("python")
        #--> 0
        ? Why()
        #--> refused: 'python' has NO declared trust posture
        DeclarePosture("python", :External)
        ? MayExecute("python")
        #--> 1
        ? Why()
        #--> allowed: posture 'external' declared
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le refus dit pourquoi : `refused: 'python' has NO declared trust posture`. Un exécuteur sans posture de confiance déclarée ne procède jamais.

Et le jumeau peut répéter un lancement avant qu'il s'exécute, signalé comme un risque. Le chapitre 24 traite du jumeau, et un bloc ici suffit pour dire ce que la porte n'utilise pas :

HARO

```
StzVirtualEnvironmentQ() {
        Spawn("python analyse.py")
        ? NumberOfPendingSpawns()
        #--> 1
        oPlan = GenerateUpdatePlan()
}
? oPlan.Narration()
#--> Update plan (1 of 1 operations to commit):
#-->   * 1. spawn process: python analyse.py
? @@( oPlan.Risks() )
#--> [ [ 1, "RUNS an external command (side effects): spawn process: python analyse.py" ] ]
```

✓ exécuté : a affiché ce que ce bloc promet

Rien n'y est exécuté, et le plan signale le lancement : `RUNS an external command (side effects): spawn process: python analyse.py`. Le script nommé n'existe pas.

Vient ensuite ce que j'ai décidé pour la porte, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé d'une seule porte. Son nom de vision est stzx, la porte des langages étrangers, le x désignant l'externe. Les noms retirés sont deux plus anciens. La lettre X a été refusée pour l'algorithmique, parce qu'elle se serait tenue à côté de la porte, la collision que le fichier propre de la vision sur les lettres de la plateforme a nommée comme la plus aiguë. La porte a pris une lettre et l'a gardée.

J'ai refusé d'autres intégrations d'hôtes. La porte est la réponse à « et le langage X ? ». Le chapitre 1 a promis ce chapitre sur ces mots : sept langages ont été choisis pour sept couches d'une solution.

J'ai refusé d'abandonner les domaines au code généré. L'article sur la bibliothèque de domaines le pose comme une position. Plutôt que de forcer les programmeurs à maîtriser chaque domaine, ou à s'abandonner au code généré par l'IA, fournir des abstractions soigneusement conçues. Le site, tel que je l'ai lu le 9 octobre 2026, dit que l'article est dans le dépôt depuis le 23 avril 2025. Son code est absent.

J'ai refusé une facette comme runtime. Le plan de l'étude le dit en une ligne : une facette est une syntaxe d'une grammaire et jamais un runtime. La porte reste donc le seul chemin vers le code étranger. C'est un plan, proposé. Ce n'est pas tranché.

J'ai refusé un afficheur qui approche. La règle de la note de conception pour les facettes : un afficheur refuse avant d'approcher. Une facette épelle le graphe et ne lui donne jamais le sens de Python ou de JavaScript.

Voyons maintenant le tribunal, et ce qu'il prouve de la porte.

5. LE TRIBUNAL

`51_extercode_narrated` prouve que le registre connaît Python, R, Julia, Prolog, C et NodeJS, et rejette un nom inconnu. Il prouve que la recherche ignore la casse (testé pour Python) ; que le code posé est le code relu ; et que la fabrique construit et transmet. Le test de fumée qu'il narre dit dans son en-tête : pas d'`Execute`, par son propre garde-fou. `governance_narrated`, scène six, prouve qu'une exécution sans posture refuse et qu'une posture déclarée procède et se nomme ; la scène sept, que les postures voyagent par la sauvegarde et le chargement. `virtual_environment_twin_narrated` prouve qu'un lancement mis en file ne s'exécute pas avant la validation, par un fichier témoin ; qu'une portée refuse ; et que le plan narre et signale le lancement. `pycodegraph_narrated` et `jscodegraph_narrated` prouvent les classes, les méthodes, les chaînes d'héritage, les propriétaires, l'impact et la cascade sur des sources Python et sur des sources JavaScript, hors ligne et sans le runtime. Tous deux sautent leurs scènes d'analyse quand le tree-sitter du moteur est absent, et le premier saute aussi celles de l'AST Python quand Python n'est pas sur le chemin.

Je veux être exact sur le tribunal de la porte. Une garde de l'instantané qui exécute un runtime étranger et passe est absente ; ce que tient aujourd'hui le tribunal de la porte, c'est un registre et un test de construction. Le propre relevé d'exécution du répertoire, sous une version plus ancienne du runtime, énumère trente-sept échecs. Plusieurs promesses ne peuvent pas correspondre par construction, parce que les noms de fichiers portent un suffixe aléatoire et les appels une durée.

Le terrain vient ensuite, et où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`51_extercode_narrated` 11 assertions
base/test/extercode/51_extercode_narrated.ring

`governance_narrated` 19 assertions
base/test/governance/governance_narrated.ring

`virtual_environment_twin_narrated` 38 assertions
base/test/system/virtual_environment_twin_narrated.ring

`pycodegraph_narrated` 48 assertions
base/test/meta/pycodegraph_narrated.ring

`jscodegraph_narrated` 13 assertions
base/test/meta/jscodegraph_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en vitesse, et la vitesse est ce pour quoi la porte existe. L'article sur la performance mesure sept runtimes sur trois tâches, et l'hôte arrive dernier dans deux d'entre elles. Je reprends ses réserves, que le chapitre 22 énonce aussi : il n'a ni versions ni dates dans son texte, et il dit que ses temps sont relatifs à un environnement informatique précis. Trier un million d'éléments prend à l'hôte environ trente secondes et à Python environ quatre, dans la table de l'article. Je n'en imprime pas davantage.

Le terrain mène dans les runtimes qu'il peut réellement exécuter. Sur six portes, deux ont tourné sur cette machine, Node et Python ; la porte R a démarré son runtime et levé une erreur ; les trois autres n'avaient aucun runtime à démarrer.

L'étude dit que sa description des outils polyglottes du terrain a été écrite d'après le genre et non relue, et je ne revendique aucune comparaison mesurée.

Le terrain mène dans l'écosystème et dans l'usage. La page de comparaison ne parle de Python que comme d'un rival ; pour couvrir ses trente lignes en Python, on assemble numpy, pandas et d'autres. La porte est la façon dont une solution Softanza les prend sans partir. Je tiens cela pour une vue, non pour un résultat : la section 3 montre la porte portant deux runtimes dans un seul pipeline, et rien dans cette page ne fait tourner numpy ou pandas.

La portée du tribunal s'arrête aujourd'hui à la porte : le processus étranger est hors de lui, et le modèle de menace du chapitre 21 met hors de portée le code déjà exécuté dans le processus.

Et le coût d'une facette, dans la table de la note : un lecteur de Python attend une indexation à partir de zéro, et celle de Haro part de un. Le littéral `\n` est une barre oblique inverse et un `n` ; deux lettres de casse différente sont une seule variable. « Tout modèle écrit Haro sous la forme qu'il connaît » n'est vrai que des graphies qui sont la minorité.

7. LA LIMITE

CONSTRUIT	un registre fermé de six runtimes étrangers et un appel par processus qui rapporte sous forme de listes les résultats de Python et de NodeJS ; les postures de confiance ; la répétition d'un lancement par le jumeau ; un lecteur de Python et de JavaScript vers un seul graphe de code
CONÇU	la porte gouvernée, avec une posture demandée à l'appel ; le pont du moteur et ses trois niveaux ; la bibliothèque de domaines au-dessus de la porte ; Haro portant une syntaxe Python et une syntaxe JavaScript
ABSENT	une porte dont le chemin du retour est fermé ; une porte pour R qui s'exécute ; une exécution réussie d'un runtime étranger dans le tribunal de la bibliothèque ; le modèle d'IA comme porte

Le mot *gouverné* est une conception. La porte interroge-t-elle la gouvernance avant de s'exécuter ? Voici le bloc qui le montre :

HARO

```
oGov2 = new stzGovernance("shop")
? oGov2.MayExecute("python")
#--> 0
oPy3 = new stzExterCode(:python)
oPy3 {
        SetRuntimePath(cPython)
        SetCode("res = 6 * 7")
        Execute()
        ? Result()
        #--> 42
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

La gouvernance a dit non, et la porte s'est exécutée quand même, et a renvoyé `42` ! Rien dans le répertoire de la porte ne mentionne la gouvernance ni la posture. La porte appelle directement le système, et non le jumeau.

Le chemin du retour n'est pas encore fermé. Le fichier de résultat est lu par le propre `eval` de l'hôte, et une chaîne qui sort du runtime étranger est enveloppée entre apostrophes simples sans échappement. Une chaîne étrangère qui porte une apostrophe et un appel peut-elle atteindre le code de l'hôte ? La fonction utilisée ici est `upper`, inoffensive exprès :

HARO

```
oPy4 = new stzExterCode(:python)
oPy4.SetRuntimePath(cPython)
oPy4.SetCode(`
res = "x' + upper('y') + 'z"
`)
oPy4.Execute()
? oPy4.Result()
#--> xYz
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le fichier de résultat contenait `'x' + upper('y') + 'z'`, et `Result()` l'a évalué. Les données sont devenues un appel, et le bloc a imprimé `xYz` ! Une simple apostrophe dans des données, `it's fine`, écrit une chaîne que l'hôte ne peut pas lire, et la méthode imprime l'erreur et renvoie le texte brut. C'est la même classe de limite ouverte que la liste du chapitre 11 dans une phrase naturelle. Je n'ai pas testé la conversion des cinq autres langages.

La porte R est en développement : aujourd'hui elle lève une erreur à chaque appel, parce que son script généré contient un identifiant qui commence par un tiret bas, que R rejette. Je l'ai confirmé avec R sur un fichier d'une ligne. Et la trace enregistre un code de sortie de 0 pour un appel qui a échoué. Le code passe un 0 littéral à l'enregistreur, et l'échec n'est détecté que par l'absence du fichier de résultat :

HARO

```
oStat = new stzExterCode(:R)
oStat {
        SetCode("res <- c(1, 2, 3)")
        bRaised = 0
        try
                Execute()
        catch
                bRaised = 1
        done
        ? bRaised
        #--> 1
        ? CallTrace()[1][:exitcode]
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

L'appel a levé une erreur, et la trace garde pourtant `0` ! Le message de R était en français sur cette machine, je ne le promets donc pas. C'est la limite de la porte à ce commit, dans la voix de la liste ouverte du chapitre 11. Les exécutions ci-dessus en sont la preuve. Gouverner la porte est conçu et pas encore construit, et le mot « gouvernée » de la vision reste donc une conception. Le chemin du retour n'est pas encore fermé, puisque les données reviennent par un fichier que l'hôte évalue. Et la porte R, en développement, lève une erreur à chaque appel aujourd'hui.

Ce qui est conçu en plus : le pont polyglotte propre du moteur, que la conception du moteur marque comme prévu. Un module de même nom dans le macroplan, un registre de chaînes, a été marqué fait alors qu'il n'avait aucune source. Puis les trois niveaux pour Python, dont seul le sous-processus externe existe. Puis la bibliothèque de domaines au-dessus de la porte, dont je n'ai trouvé aucune classe dans le code. Et les facettes, une note qui propose et ne tranche rien.

Ce qui est absent : les fonctions de l'article qui font entrer les données, `@@R`, `@@C`, `XC()` et `XAI()`, qui ne sont définies dans aucun fichier. Des fabriques existent pour Python, R, Julia et Prolog. Les portes C et NodeJS ne sont atteintes que par la classe. Le modèle d'IA n'est pas du tout une porte de ce volet ; la fonction de l'article qui l'exécute dans l'hôte est le contraire de la loi du chapitre 17, et le livre ne l'enseigne pas.

Et une forme de porte que je n'ai pas exécutée : Julia, C et Prolog, pour lesquels aucun runtime n'est installé ici. Et une troisième relation, nommée dans l'étude à côté de la porte : l'adaptateur qui colle une solution trouvée sur internet dans un autre langage et la change un peu. `range0`, le range de Python comme fonction de l'hôte, a une garde. Le relevé propre du dossier lui compte cinq échecs sur des plages qui devraient être vides et un sur la forme de tranche `':5:-1'`. Je n'en ai rien exécuté.

Six lectures de la porte suivent, une pour chacun de ceux qui la rencontrent.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous écrivez du Haro et, pour la couche qu'un autre langage fait le mieux, vous lui remettez une chaîne de code et une liste de données et vous reprenez une liste. Dans le plan, un programmeur Python ou JavaScript écrira aussi du Haro dans une syntaxe qu'il connaît. Deux choses différentes : une syntaxe vous amène, et un runtime amène vos bibliothèques.
L'analyste	La couche statistique est celle de R, la couche d'apprentissage automatique celle de Python, la couche logique celle de Prolog. Vous restez dans un fichier et une liste de résultats. La porte R est en développement et lève une erreur aujourd'hui, et le dossier le dit.
Le concepteur	Il n'y a rien à concevoir à la porte elle-même. La leçon du dossier est son refus : un langage hors de la liste est refusé par son nom, et « et mon langage ? » a une seule réponse.
Le décideur	La porte est l'unique endroit où la fermeture prend fin, par décision. À ce commit sa gouvernance est conçue et pas encore construite, et son chemin du retour, pas encore fermé, atteint le code de l'hôte. Le mot « gouvernée » de la vision est une conception. La durée de chaque appel est enregistrée aujourd'hui dans la trace.
Le formateur	Un apprenant voit quelle couche appartient à quel langage et exécute un pipeline à deux portes. La leçon est la différence entre lire un langage, l'exécuter, et écrire dans sa syntaxe.
L'agent	Vous n'atteignez jamais la porte par vous-même. Une fonction dans le fichier d'un agent d'IA doit porter une posture, et `external` convient à un runtime étranger ; et un modèle d'IA n'a aucune capacité de produire des effets (chapitre 20).

9. EXERCICE

Vous construisez une petite solution avec trois tâches : trier un million de nombres, décider lequel de deux étudiants réussit, et vérifier dans un arbre généalogique qui est l'ancêtre de qui. Pour chaque tâche, choisissez dans la liste des couches de la section 3 le langage qui possède sa couche, et écrivez votre raison en une ligne. Puis, sans rien exécuter, écrivez la promesse pour chaque ligne de ce bloc, et exécutez-le :

HARO

```
oDoor = new stzExterCode(:python)
# promise: the door answers "julia" with ...
# promise: the door answers "ring" with ...
# promise: asking for "cobol" raises with the sentence ...
oGov = new stzGovernance("mine")
# promise: oGov.MayExecute("prolog") before any posture is declared gives ...
# promise: after oGov.DeclarePosture("prolog", :External), it gives ...
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

La dernière promesse est celle qui compte. Si vous appelez maintenant la porte pour Prolog, la gouvernance aura-t-elle été interrogée ? Écrivez oui ou non, et pourquoi. Puis lisez la section 7. Si une promesse se brise, ne la changez pas : expliquez d'abord la sortie.

Je vous laisse celui-ci. La section 7 dit qu'une simple apostrophe dans des données fait écrire au chemin du retour une chaîne que l'hôte ne peut pas lire, et je n'ai imprimé aucune sortie pour elle. Voici le bloc de la section 7 avec un seul changement, `it's fine` comme chaîne étrangère :

HARO

```
oPy4 = new stzExterCode(:python)
oPy4.SetRuntimePath(cPython)
oPy4.SetCode(`
res = "it's fine"
`)
oPy4.Execute()
? oPy4.Result()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Lisez ce que la méthode imprime et ce qu'elle renvoie, et expliquez les deux avant de toucher au bloc ; la page dit pourquoi dans la section 7, là où le chemin du retour n'est pas fermé. Quand vous le pourrez, vous saurez pourquoi un chemin du retour qui évalue ses données n'est pas fermé.

Le chapitre 17 ouvre la partie sur l'intelligence par sa loi : programmatique par défaut, le modèle en dernier, l'admission plutôt que la génération. Un modèle d'IA y est une faculté parmi d'autres, la dernière appelée, et ce qu'il dit est une proposition

que le tribunal admet ou refuse.

OÙ SONT LES PREUVES

NARRATION	`stzexcis-polyglot-programming-in-ring.md`
NARRATION	`stzexterlib-domain-driven-polyglot-programming.md`
NARRATION	`stzexcis-performance-battle.md`
NARRATION	`stz-pycode-bridging-ring-and-python-narration.md`
SUR LE SITE	architecture
DÉCISIONS DE L'ÉTUDE	12.19, 12.20, 12.21, 12.22, 12.28, 12.29

PARTIE IV

L'intelligence, repensée

> *Programmatique par défaut, le modèle en dernier, l'admission avant la génération.*

17 · Loi : programmatique par défaut, le modèle en dernier, l'admission plutôt que la génération
18 · Les cinq registres, et les conversations qui demandent
19 · Le modèle de langage de domaine
20 · Des agents qui ne peuvent pas vous nuire
21 · Le tribunal : promesses, perception, sécurité et performance
22 · Comparé, honnêtement

PARTIE IV · CHAPITRE 17 · COLONNE

Loi : programmatique par défaut, le modèle en dernier, l'admission plutôt que la génération

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 8 bloc(s) sur 8 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, j'énonce la loi de la partie IV et je la fais tourner. Je commence par ma définition de l'intelligence et par un modèle de langage de domaine sans aucun neurone, qui répond, dit pourquoi, et refuse ce qu'on ne lui a jamais dit. Puis je montre une conversation qui vous interroge, un fait que personne n'a enregistré prouvé à partir d'une seule loi déclarée, un modèle d'IA tenu à proposer, et un agent dont l'esprit est du code, refusé à la barrière. Je dis ce que j'ai refusé, quelles gardes prouvent chaque bloc, et où le terrain mène. Vient ensuite la limite, imprimée comme l'étape qu'elle est : une question posée avec d'autres mots, et une réponse là où un refus est l'étape suivante. La loi elle-même tient en une phrase : l'esprit est programmatique par défaut, un modèle est une faculté parmi d'autres, la dernière appelée, et ce qu'il dit est une proposition que le tribunal admet ou refuse.

CE QUE L'ÂGE DE L'IA CHANGE ICI

Dans les deux mouvements qui précèdent celui-ci, l'âge de l'IA a changé une chose chacun. En programmation, il a changé qui écrit le code. Dans les langages, il a changé qui prononce la phrase. Ici il change tout, parce que ce mouvement porte sur l'intelligence elle-même, et que le terrain emploie désormais le mot surtout pour dire une seule chose : un grand modèle de langage à l'intérieur de votre logiciel.

Alors ce chapitre fait ce que font les deux autres chapitres de loi, et énonce la loi de sa partie avant les preuves. Il dit aussi comment la partie dépend des autres, parce que l'ordre de ce livre va du familier au nouveau. L'intelligence, dans Softanza, est gouvernée par ce qui l'a précédée. Le tribunal qui juge la phrase d'un modèle est le tribunal de la Partie II, qui juge le code quel que soit son auteur. La phrase qu'il juge est une phrase d'un langage fermé, qui est la Partie III. On ne peut mettre un modèle en dernier que parce qu'il y a quelque chose de solide à mettre en premier.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Demandez au terrain ce qui rend une application intelligente et la réponse est un modèle d'IA. Un agent d'IA, au sens du terrain, est un modèle dans une boucle avec des outils, une mémoire et un but. La liste de courses pour en construire un en 2026 est longue. Il faut un projet en Python, un fournisseur de modèle, un framework qui câble la boucle comme un graphe, un magasin de vecteurs pour la mémoire, des outils avec des schémas typés, un service de traçage, et un chapitre de conseils de sécurité. La mémoire est souvent de la récupération : une collection de documents, découpés en morceaux, cherchés par similarité et collés dans le prompt.

La sécurité, dans ce tableau, est une propriété de l'agent. C'est l'alignement, les garde-fous, un prompt qui demande confirmation, une case marquée « humain dans la boucle ». Le terrain a aussi appris à tenir la sortie d'un modèle dans une forme, avec des grammaires qui contraignent ce qu'il peut émettre. Tout cela est un travail sérieux, et une partie est bonne. Tout cela partage une hypothèse : le modèle est l'esprit, et tout le reste est autour de lui.

2. LE PREMIER PRINCIPE

L'esprit est programmatique par défaut ; un modèle est une faculté parmi d'autres, la dernière appelée, et ce qu'il dit est une proposition que le tribunal admet ou refuse.

3. LA FORME SOFTANZA

Commençons par ma définition, parce que tout ce qui suit en découle. L'intelligence est la capacité computationnelle de dériver de nouvelles connaissances à partir de connaissances existantes. Rien dans cette phrase n'a besoin d'un modèle.

Voici un modèle de langage de domaine sans aucun neurone, forgé à partir d'un petit graphe de connaissances d'un restaurant. Que répond-il à partir de ce qui a été enregistré, et que dit-il de la coupe du monde de 1998 ? Exécutez-le :

HARO

```
oKB = new stzKnowledgeGraph("restaurant")
oKB {
        Know("margherita", "dish")
        KnowRelation("margherita", "contains", "tomato-sauce")
        KnowRelation("margherita", "contains", "mozzarella")
}
oDLM = StzDlmQ(oKB)
oDLM {
        ? Ask("margherita contains what?")
        #--> Margherita contains tomato-sauce and mozzarella.
        ? Why()
        #--> recorded fact(s): 'margherita' contains tomato-sauce, mozzarella
        ? AskXT("margherita contains what?")[:certainty]
        #--> 1
        ? Ask("who won the 1998 world cup?")
        #--> That is outside the 'restaurant' domain (I know 1 entities and 1 relations).
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond, dit pourquoi (`recorded fact(s): 'margherita' contains tomato-sauce, mozzarella`), et énonce sa certitude (`1`). Et quand on lui demande ce qu'il ne sait pas, il ne devine pas. Il refuse, et il vous dit combien son monde est petit : `I know 1 entities and 1 relations`. Une entité, parce que seule `margherita` a été déclarée par `Know`, et une relation, `contains` : les objets des deux faits ne sont pas comptés comme des entités. Un refus qui nomme sa frontière vaut mieux qu'une réponse fluide qui n'en a pas. C'est mon point de vue.

Voici maintenant la deuxième pièce, celle qui retourne le dialogue habituel. Sur le terrain, vous soumettez un prompt à la machine et gardez ce qui en sort. Ici le système vous interroge. Que demande-t-il, et que se passe-t-il si vous essayez de conclure avant d'avoir répondu ?

HARO

```
StzKnowledgeGraphQ("menu") {
        Know("margherita", "dish")
        Know("tiramisu", "dish")
        AddConversationQ("setup").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        ? AskIn("setup")
        #--> What does 'margherita' have for 'contains'?  (why: every dish needs 'contains')
        try
                ConcludeIn("setup", "menu")
        catch
                ? cCatchError
        done
        #--> Can't conclude: 2 gap(s) remain -- the wise-coding loop is not done. Ask the next
question.
}
```

✓ exécuté : a affiché ce que ce bloc promet

Un but a été déclaré : chaque plat doit dire ce qu'il contient. L'écart entre le but et ce qui est connu a engendré la question, et la question porte sa raison. Le monde ne peut pas être clos tant qu'il est incomplet : `Can't conclude: 2 gap(s) remain`. Le chapitre 18 montre les cinq façons de répondre : une option d'une liste, une structure de données, une formule, une phrase, ou des exemples. Trois sont câblées ; la formule et les exemples sont absents. Un modèle d'IA, s'il y en a un, vient après toutes, pas en premier.

La troisième pièce est la dérivation, ma définition à l'œuvre. Une loi est déclarée, et un fait que personne n'a enregistré est prouvé :

HARO

```
StzKnowledgeGraphQ("kitchen") {
        ConstrainRelation("contains", :Transitive)
        KnowRelation("margherita", "contains", "tomato-sauce").
            KnowRelation("tomato-sauce", "contains", "tomato")
        ? Prove([ "margherita", "contains", "tomato" ])[:narration]
        #--> proved: margherita contains tomato-sauce contains tomato
        ? Prove([ "margherita", "contains", "luigi" ])[:narration]
        #--> not derivable: no recorded fact or lawful chain gives 'margherita' contains 'luigi'
}
```

✓ exécuté : a affiché ce que ce bloc promet

De la connaissance nouvelle à partir de la connaissance existante, avec sa chaîne écrite en toutes lettres ! Et un « non » aussi certain que le « oui », parce qu'il est le résultat d'une recherche, pas d'une humeur. La page l'a imprimé ainsi : `not derivable: no recorded fact or lawful chain gives 'margherita' contains 'luigi'`. Il est certain de ce qui est enregistré et déclaré, pas du monde : il dit `not derivable`, pas « faux ». Pas de récupération, pas d'entraînement. Ajouter un fait fait que le système sait davantage ; ajouter une loi lui fait raisonner plus loin.

Puis le modèle d'IA. Il est le bienvenu, et il est tenu. Branchez la sortie d'un modèle directement sur un effet, ici l'envoi d'un message. La composition est-elle saine ?

HARO

```
oG = new stzAgentGraph("mailer")
oG {
        AddLLMActor("writer")
        AddEffect("send")
        Proposes("writer", "send")
        ? IsSound()
        #--> 0
        ? len(Violations())
        #--> 3
}
```

✓ exécuté : a affiché ce que ce bloc promet

Elle ne l'est pas : `IsSound()` a imprimé `0`, et elle est refusée avant que rien ne s'exécute ! Trois lois se déclenchent, et `len(Violations())` a imprimé `3` : tout effet doit passer une barrière, tout texte libre doit avoir un endroit sûr où atterrir, et tout effet doit laisser un témoin. Que détient un acteur modèle qui puisse agir ? Aucun modèle d'IA n'est chargé dans le bloc qui le demande : la réponse est fixée d'avance, comme un test fixe une réponse golden, afin que la page s'exécute de la même façon chaque nuit. Interrogez-en un :

HARO

```
oF = new stzLLMFunction("sum")
oF {
        SetPrompt("summarize: {input}")
        Budget(2)
        SeedAnswer("hello world", "a greeting")
```

```
}
oLLM = new stzLLMAgent("summarizer")
oLLM {
        SetSkillFrom(oF)
        ? Propose("hello world")
        #--> a greeting
        ? HoldsEffectful()
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

Rien. Un acteur modèle ne détient aucune capacité à effet ; il propose, et c'est tout son pouvoir. `HoldsEffectful()` a imprimé `0`. Ce que le bloc prouve ne dépend pas du modèle : quoi qu'il dise, il ne détient rien qui puisse agir.

Et l'agent programmatique lui-même a un esprit fait de code. Une compétence est une condition et une action, et elle se déclenche sur ce que l'agent perçoit. La gouvernance soumet l'acte à la barrière avant qu'il ait lieu. Quand le stock est bas et que la compétence veut commander, l'acte passe-t-il ?

HARO

```
oRog = new stzPIAgent("rogue-bot")
oRog.MemoryQ().Learn("stock", "level", "low")
oRog.GovernanceQ().DeclareRisk("order-stock", 2)
oSk = new stzAgentSkill("restock")
oSk.SetWhen(func oM { return oM.Fact("stock", "level", "low") })
oSk.SetDoes(func oM { oM.Learn("stock", "level", "ordered")  return 1 })
oRog.AddGovernedSkill(oSk, "order-stock")
? oRog.Pursue()
#--> 0
? oRog.DecisionLog()[1][:why]
#--> refused: 'rogue-bot' lacks PERMISSION (can) for 'order-stock'
```

✓ exécuté : a affiché ce que ce bloc promet

Le stock est bas, la compétence veut commander, et l'agent n'a jamais reçu la permission. `Pursue()` a imprimé `0` : la gouvernance l'a refusé ! Il ne se passe rien, et le journal dit pourquoi, par son nom : `refused: 'rogue-bot' lacks PERMISSION (can) for 'order-stock'`. Le monde est comme il était.

Mettez les pièces sur une ligne et vous avez la pile que la vision de la plateforme nomme son Étoile polaire, ratifiée le 27 septembre 2026 : percevoir, décider, réagir, apprendre. Rien de tout cela n'a besoin d'un modèle de langage. C'est une conception, pas un agent assemblé. Dans cette conception, la perception classe le monde dans le graphe de connaissances. Un planificateur décide. Des règles réagissent. L'agent note ses plans et garde le meilleur. Les pièces existent dans la bibliothèque à côté de l'agent, pas encore dedans : l'agent ci-dessus n'en appelle aucune. Un modèle peut se tenir à côté de chaque étape comme une faculté de plus, et aucune ne dépend de lui.

Le chapitre 18 raconte les cinq registres d'une réponse et la conversation qui demande. Le chapitre 19 raconte le modèle de langage de domaine, de son premier échelon, sans neurones, à son second, entraîné sur les faits propres du premier et rien d'autre. Le chapitre 20 raconte l'agent qui ne peut pas vous nuire : il propose, un atelier répète, le tribunal juge, et seul un acteur gouverné valide. Le chapitre 21 raconte le tribunal lui-même. Le chapitre 22 compare tout cela au terrain, et imprime où le terrain mène.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé le modèle d'IA comme esprit. Quand les modèles sont arrivés, le geste était d'en mettre un au centre et de l'enclore. J'ai écrit le contraire dans l'architecture : l'esprit est programmatique par défaut, et un modèle est une faculté enfichable, l'une parmi plusieurs, jamais la définition de l'esprit. La boucle de l'industrie est un cas particulier de cette conception, avec une seule faculté et aucune gouvernance. La plus ancienne phrase de moi que le dossier tienne là-dessus est l'architecture de juillet 2026.

J'ai refusé tout ce qui est payant ou distant comme exigence. Chaque capacité a un plancher qui n'exige ni installation, ni abonnement, ni réseau. Un modèle peut être ajouté par-dessus. Il n'est jamais le prix d'entrée. « En dernier » veut dire après le chemin programmatique, et sous le tribunal et un budget. Que le dernier proposant puisse être un modèle distant sous ce budget, ou doive toujours être un modèle local, le dossier admet les deux lectures, et je ne tranche pas ici entre elles.

J'ai refusé de deviner. Toute ambiguïté refuse plutôt que de deviner. L'habitude du terrain de soumettre un prompt et de garder ce qui sort a désormais un nom ; ici, deviner est remplacé par demander, et les impressions sont remplacées par la gouvernance.

J'ai refusé la récupération comme mémoire. La mémoire est un graphe gouverné, non un tas de fragments semblables. La dérivation remplace la récupération : le système sait ce qu'on lui a dit et ce qui en découle par la loi, et il peut prouver les deux.

J'ai refusé un enseignant que je ne peux pas voir. Le second échelon du modèle de langage de domaine est entraîné sur les faits de la base de connaissances du premier échelon et sur rien d'autre. Aucun modèle d'IA distant ne l'enseigne. La conception va plus loin et dit que le texte d'entraînement du second échelon est validé par la grammaire du premier avant d'être admis, de sorte qu'il est correct par

construction. Dans la bibliothèque, les faits sont rendus directement en texte, sans encore une telle vérification ; le chapitre 19 dit où cela en est. J'ai donné le concept d'un modèle de langage de domaine, conçu d'abord pour l'un des produits du domaine de projets, à la fondation ouverte. Tout projet qui fournit une base de connaissances peut livrer gratuitement son propre modèle.

J'ai refusé un bouchon. La bibliothèque déclare cinq agents propres et en a câblé un. En demander un qui n'est pas câblé lève une erreur ; elle ne renvoie pas quelque chose qui ressemble à un agent. Faire semblant serait plus facile, et cela ferait valoir moins tous les autres refus de ce chapitre.

Derrière les blocs se tiennent six gardes, et la section suivante les nomme.

5. LE TRIBUNAL

Chaque bloc ci-dessus repose sur une garde. `dlm_narrated` prouve le modèle de langage de domaine : des réponses avec leur pourquoi, le refus hors du domaine, la complétion tenue aux entités et aux relations du domaine, un lot autonome, et un second échelon entraîné sur son propre corpus. `wisecoding_narrated` prouve la conversation qui demande et le refus de conclure. `knowledge_integration_narrated` prouve la dérivation et le « non » certain. `agentgraph_narrated` prouve qu'un modèle branché directement sur un effet n'est pas sain, que la forme gouvernée l'est, et qu'un acteur modèle propose et ne valide jamais. `piagent_narrated` prouve l'agent programmatique et son refus à la barrière. `ownagentstack_narrated` prouve la liste honnête des agents propres de la bibliothèque.

Ces six gardes prouvent ce qui a tourné ; la section suivante dit où le terrain mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`dlm_narrated` 16 assertions base/test/learning/dlm_narrated.ring

`wisecoding_narrated` 13 assertions
base/test/conversation/wisecoding_narrated.ring

`knowledge_integration_narrated` 19 assertions
base/test/natural/knowledge_integration_narrated.ring

`agentgraph_narrated` 11 assertions
base/test/agentic/agentgraph_narrated.ring

`piagent_narrated` 14 assertions
base/test/agentic/piagent_narrated.ring

`ownagentstack_narrated` 10 assertions
base/test/agentic/ownagentstack_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en portée. Ses frameworks se connectent à des services hébergés de toutes sortes, ses magasins cherchent dans des collections ouvertes de documents, et ses boucles appellent les plus grands modèles disponibles à chaque étape. L'inversion de Softanza, constituer un domaine et en dériver, mène sur un domaine constitué, et sur le web ouvert le terrain mène aujourd'hui. Quand la tâche est un travail de langage ouvert sur une matière que personne ne peut constituer, résumer des documents arbitraires ou faire de la recherche sur le web, la pile du terrain est le bon outil. Le dossier le dit en ces mots.

Le terrain mène sur le marché. Il existe aujourd'hui un marché du travail pour sa pile ; celui de cette pile est à venir.

Le terrain mène sur les modèles d'IA. Le second échelon du modèle de langage de domaine, tel que construit, est un petit modèle neuronal de paires de mots, et la fonderie au-delà du bigramme est conçue (chapitre 19). Softanza ne construit pas les grands générateurs et ne rivalise pas avec eux.

Là où je pense que la loi se distingue, c'est en trois endroits. Une décision ici ne coûte rien, s'exécute hors ligne et donne la même réponse chaque jour, de sorte qu'on peut la tester avec des promesses ordinaires. Un modèle d'IA ici ne peut pas agir, par construction, pas par vigilance. Et la phrase d'un modèle, là où l'on en emploie un, peut être tenue à la grammaire d'un langage déclaré avant d'être émise, au lieu d'être vérifiée puis relancée après.

La section suivante dit où la loi s'arrête.

7. LA LIMITE

CONSTRUIT — le premier échelon du modèle de langage de domaine et son second, un petit bigramme ; la conversation qui demande ; la dérivation avec sa preuve ; l'agent programmatique ; la barrière qui refuse un modèle d'IA non gouverné

CONÇU — la cascade de la machine pensante entre les facultés ; la pile de décision, ses pièces à côté de l'agent et pas encore dedans ; l'unité typée de prédiction, une conception du domaine de projets avec cinq étapes de validation et aucune chaîne de celles-ci dans la bibliothèque ; la négociation entre un agent et son relecteur

ABSENT — quatre des cinq agents propres de la bibliothèque, dont l'induction à partir d'exemples

Ce qui est construit : le premier échelon du modèle de langage de domaine et son second, un petit bigramme, la conversation qui demande, la dérivation avec sa preuve, l'agent programmatique, et la barrière qui refuse une composition non gouvernée.

Ce qui est conçu : la machine pensante du chapitre 9, où de nombreuses facultés partagent un état et se déclenchent en cascade jusqu'à ce qu'il se stabilise. La pile de décision qui perçoit, décide, réagit et apprend, dont les pièces existent à côté de l'agent et pas encore dedans. L'unité typée de prédiction, que le chapitre 19 raconte comme une conception du domaine de projets, validée en cinq étapes dans cette conception et enchaînée nulle part dans la bibliothèque. La conversation entre un agent et celui qui relit son plan : le plan atteint désormais l'agent, et la conversation à son sujet pas encore.

Ce qui est absent : quatre des cinq agents propres de la bibliothèque. Combien la bibliothèque en déclare-t-elle, et combien sont câblés ? La liste le dit clairement :

HARO

```
? len(StzOwnAgents())
#--> 5
? StzOwnAgentIsWired("wise-coder")
#--> 1
? StzOwnAgentIsWired("induction")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Le validateur, le classeur, le planificateur et l'agent qui induirait un motif à partir de vos exemples sont déclarés et réservés. Ainsi le cinquième registre d'une réponse, les exemples, attend un agent qui n'existe pas encore.

Et une limite du premier échelon que vous rencontrerez dès que vous reformulerez une question. Il ne reconnaît une relation que par le mot avec lequel elle a été déclarée. Que fait-il quand vous demandez avec d'autres mots ?

HARO

```
? oDLM.Ask("what does margherita contain?")
#--> Margherita is a dish.
```

✓ exécuté : a affiché ce que ce bloc promet

Il n'a pas refusé ! Il a imprimé `Margherita is a dish.` Cette réponse est vraie, et ce n'est pas ce qui était demandé. Par la loi de ce chapitre, la réponse ici est un refus, et ce refus est l'étape suivante. Je l'imprime ici parce qu'une loi se montre par une page qui imprime son étape telle qu'elle est, limite comprise. La demande à la bibliothèque est de refuser une question dont le verbe n'est pas une relation déclarée.

8. SIX PORTES

Voici maintenant l'exercice : un domaine à vous, et les promesses que vous attendez.

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	L'esprit de votre agent est du code que vous pouvez tester avec des promesses ordinaires. Un modèle d'IA peut être une faculté ; il ne peut jamais détenir un effet.
L'analyste	Un fait ou une règle que vous ajoutez fait que le système en sait plus, sans changement de code et sans entraînement, et chaque réponse porte son pourquoi.
Le concepteur	Le système pose à l'utilisateur une question responsable à la fois, et la question est remise à n'importe quelle porte sous forme de cadre, pour qu'un terminal, une page ou un agent puisse la rendre. Seul le terminal a tourné ici.
Le décideur	Une décision ne coûte rien par appel, s'exécute hors ligne et est la même demain. Les quatre agents encore réservés sont l'étape suivante.
Le formateur	La conversation demande et ne répond jamais la première, et elle n'emploie la fluidité d'un modèle que lorsqu'un modèle est là, de sorte qu'un apprenant voit toujours le plancher.
L'agent	Vous pouvez proposer. Vous ne pouvez pas valider. Un refus nomme sa loi, et vous pouvez apprendre la plateforme depuis la plateforme et être corrigé par elle.

9. EXERCICE

Construisez un domaine à vous avec trois faits : une boutique, une clinique, une classe. Avant d'exécuter quoi que ce soit, écrivez sous chaque ligne la promesse que vous attendez : une question dans le domaine, une hors du domaine. Ajoutez un but selon lequel chaque élément doit avoir une relation, et écrivez la première question que le système vous posera. Puis exécutez-le.

Si une promesse est brisée, ne changez pas la promesse pour qu'elle corresponde à la sortie. Expliquez d'abord la sortie.

Je vous laisse celui-ci, volontairement. La section 7 montre une question à laquelle le premier échelon a répondu alors que la loi veut un refus, et je ne l'ai pas corrigé. Reprenez votre propre domaine et posez de nouveau l'une de vos questions promises avec d'autres mots, et un verbe que vous n'avez pas déclaré comme relation. Regardez ce qui revient et comparez-le à ce que vous avez demandé. Puis dites en une phrase pourquoi il a répondu, et quelles lignes de la section 7 vous le disent.

Quand vous l'aurez résolu, vous saurez pourquoi un premier échelon qui ne reconnaît une relation que par le mot avec lequel elle a été déclarée peut répondre à une question que vous n'avez pas posée.

Le chapitre 18 est *Les cinq registres, et les conversations qui demandent*. Son principe : une réponse peut venir sous cinq formes, et quelle que soit la forme, elle n'est qu'un candidat, et ce sont les règles du monde qui décident si elle devient connaissance.

> OÙ SONT LES PREUVES
>
> | | |
> |---|---|
> | NARRATION | `stz-agents-that-cannot-hurt-you-narration.md` |
> | NARRATION | `stz-service-virtualization-code-first-subscribe-later.md` |
> | NARRATION | `stz-knowledge-programming-narration.md` |
> | SUR LE SITE | agentic |
> | SUR LE SITE | wise |
> | DÉCISIONS DE L'ÉTUDE | 12.18, 12.19, 12.25, 12.26, 12.29 |

PARTIE IV · CHAPITRE 18

Les cinq registres, et les conversations qui demandent

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 15 bloc(s) sur 15 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je vous montre les cinq formes qu'une réponse peut prendre, une option, une structure de données, une formule, une phrase et des exemples. Puis je montre ce que la porte gouvernée fait de chacune. Je vous dis quelles sont les trois qui s'exécutent et les deux qui ne s'exécutent pas, et ce que devient un refus une fois écrit : il attend une personne, avec son contexte et une durée de vie. Je laisse s'exécuter la conversation où il n'y a personne, et je montre ce qu'une conversation peut et ne peut pas faire en arabe et en haoussa. Quelle que soit la forme, une réponse n'est qu'un candidat, et ce sont les règles du monde qui décident si elle devient connaissance.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Deux pratiques se rencontrent dans l'idée d'une conversation qui demande, et une troisième se tient derrière son dernier registre. Toute cette section est de la connaissance générale, non une étude du dossier.

La première est le remplissage d'emplacements (slot filling). Un cadre de dialogue demande à l'utilisateur les éléments manquants d'un formulaire jusqu'à ce que le formulaire soit plein. Les frameworks du terrain comprennent le texte libre avec des modèles entraînés, parlent sur de nombreux canaux et livrent des connecteurs, et ils demandent au concepteur d'écrire des intentions et des flux dans leur propre langage. Ils prennent la réponse de l'utilisateur pour la réponse et agissent dessus.

La deuxième est le vibe coding, un nom mis en circulation en 2025 : l'humain soumet un prompt, le modèle d'IA produit, l'humain garde ce qui fonctionne.

La troisième est la programmation par l'exemple. Les tableurs remplissent une colonne à partir de deux exemples que vous avez tapés. La recherche appelle cela la synthèse inductive, et l'ère des modèles a fait de l'exemple la façon de piloter un modèle.

2. LE PREMIER PRINCIPE

Une réponse peut venir sous cinq formes. Quelle que soit la forme, elle n'est qu'un candidat, et ce sont les règles du monde qui décident si elle devient connaissance.

Le document de conception le dit en une phrase que je garde : l'expression est libre, l'admission est gouvernée. Le dossier tient cette phrase, captée à partir de ma directive du 13 juillet 2026. Il ne tient aucune phrase de moi sur le pourquoi de ces cinq registres et pas quatre ou six, ni sur le moment où j'ai opposé le wise coding au vibe coding.

3. LA FORME SOFTANZA

Le chapitre 17 a montré une conversation qui demande, et qui refuse de conclure tant qu'un écart est ouvert. Ce chapitre porte sur la réponse. Les cinq registres sont cinq formes qu'une réponse peut prendre : une option choisie par son numéro, une structure de données, une formule, une phrase, et des exemples. Je vous montrerai chacun, et je vous dirai ceux qui ne s'exécutent pas.

Le premier registre : une option. Quand le monde connaît déjà des valeurs, comment la question les offre-t-elle ? Exécutez-le :

HARO

```
oKB = new stzKnowledgeGraph("pizzeria")
oKB {
        Know("pizza1", "dish")
        Know("pizza2", "dish")
        Know("pizza3", "dish")
        Know("pizza4", "dish")
        AddConversationQ("frame").SetGoal(StzGoalQ().RequireEach("dish", "topping"))
        AskIn("frame")
```

```
        ReplyIn("frame", "cheese")
        AskIn("frame")
        ReplyIn("frame", "olives")
        ? AskIn("frame")
        #--> Which 'topping' does 'pizza3' have? (1) cheese  (2) olives -- or answer freely.
(why: every dish needs 'topping')
        ? @@(ReplyIn("frame", 2)[:admitted])
        #--> [ "olives" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

La question offre les valeurs connues sous forme de numéros, et `2` a choisi `olives`.

Que répond la porte à un numéro que personne n'a offert ? Exécutez-le :

HARO

```
oKB {
        ? AskIn("frame")
        #--> Which 'topping' does 'pizza4' have? (1) cheese  (2) olives -- or answer freely.
(why: every dish needs 'topping')
        aV = ReplyIn("frame", 99)
        ? @@(aV[:admitted])
        #--> [ ]
        ? @@(aV[:refused])
        #--> [ [ "(99)", "no such option" ] ]
        ? ConversationQ("frame").Checkpoints()[1][:why]
        #--> no option (99) was offered -- 2 option(s) on the table
        ? aV[:goalState]
        #--> pursuing
}
```

✓ exécuté : a affiché ce que ce bloc promet

La porte l'a refusé, et a dit `no such option` ! Le point de contrôle dit pourquoi : `no option (99) was offered -- 2 option(s) on the table`. Une raison, et un compte des options sur la table. Il ne les énumère pas, et la question suivante offre de nouveau les mêmes options : `(1) cheese  (2) olives`. Le refus ne devine jamais une valeur, il arrive à une personne comme un point de contrôle, et le but reste `pursuing`. Cela compte pour la section 7.

La question est une donnée avant d'être un texte. Elle porte une force, qui est *what* tant que le monde ne sait rien et *which* dès qu'il connaît une valeur, et les options. N'importe quel médium peut la rendre :

HARO

```
oKF = new stzKnowledgeGraph("pizzeria2")
oKF {
        Know("p1", "dish")
        Know("p2", "dish")
        AddConversationQ("f").SetGoal(StzGoalQ().RequireEach("dish", "topping"))
        ? AskInXT("f")[:force]
        #--> what
        ReplyIn("f", "cheese")
        aQ = AskInXT("f")
        ? aQ[:force]
        #--> which
        ? @@(aQ[:options])
        #--> [ "cheese" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le deuxième registre : une structure de données. Que devient une liste de deux valeurs envoyée comme une seule réponse ? Exécutez-le :

HARO

```
oKD = new stzKnowledgeGraph("desserts")
oKD {
        Know("tiramisu", "dish")
        AddConversationQ("setup").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        AskIn("setup")
        aV = ReplyIn("setup", [ "mascarpone", "espresso" ])
        ? @@(aV[:admitted])
        #--> [ "mascarpone", "espresso" ]
        ? aV[:goalState]
        #--> fulfilled
}
```

✓ exécuté : a affiché ce que ce bloc promet

Une liste de valeurs est prise comme les valeurs, et chacune passe par la même porte.

Le quatrième registre : une phrase. Je saute le troisième un moment, parce que c'est l'un des deux qui ne s'exécutent pas. Comment une expression de la forme « X and Y » ou « X, Y » est-elle lue ? La conversation tient une transcription où chaque admission est narrée et où l'accomplissement du but est consigné :

HARO

```
oKM = new stzKnowledgeGraph("menu")
oKM {
        Know("margherita", "dish")
        AddConversationQ("setup").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        AskIn("setup")
        ReplyIn("setup", "tomato-sauce and mozzarella")
        ? ConversationQ("setup").Transcript()
        #--> SOFTANZA: Goal adopted -- this conversation is now accountable for it.
        #--> SOFTANZA: What does 'margherita' have for 'contains'?  (why: every dish needs
'contains')
        #--> YOU:      tomato-sauce and mozzarella
        #-->   -> admitted: 'margherita' contains 'tomato-sauce'
        #-->   -> admitted: 'margherita' contains 'mozzarella'
        #--> SOFTANZA: Goal fulfilled: every required slot is filled in 'menu'
}
```

✓ exécuté : a affiché ce que ce bloc promet

Elle est lue comme deux valeurs, chacune jugée : la transcription tient `admitted: 'margherita' contains 'tomato-sauce'` et `admitted: 'margherita' contains 'mozzarella'`. « Naturel » signifie ici un séparateur sur les virgules et sur le mot anglais « and ». La couche naturelle de la bibliothèque, que j'utiliserai dans un instant, n'est pas appelée par la conversation.

Quand une réponse porte plusieurs valeurs, et que le monde en permet une. Que fait une relation déclarée unique d'une réponse qui porte deux valeurs ? Exécutez-le :

HARO

```
oKS = new stzKnowledgeGraph("staffing")
oKS {
        AddConversationQ("s").SetGoal(StzGoalQ().RequireOne("kitchen3", "led-by"))
        ConstrainRelation("led-by", :Unique)
        AskIn("s")
        aV = ReplyIn("s", "mario and giovanni")
```

```
        ? @@(aV[:admitted])
        #--> [ "mario" ]
        ? @@(aV[:refused])
        #--> [ [ "giovanni", "'kitchen3' already bears 'led-by' (to 'mario'); the relation is
:Unique" ] ]
        ? len(ConversationQ("s").Checkpoints())
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

La première valeur a été admise, et la seconde refusée avec la raison de la loi : `'kitchen3' already bears 'led-by' (to 'mario'); the relation is :Unique`. Le refus arrive à une personne, comme un seul point de contrôle. Prenez garde à ce que vous lisez là-dedans. Le choix entre Mario et Giovanni a été fait par l'ordre. Il n'a pas été fait par une personne. La conception promet autre chose, et la section 7 dit quoi.

Le troisième registre, la formule. Une formule tapée en réponse est-elle évaluée ? Exécutez-le :

HARO

```
oKP = new stzKnowledgeGraph("shop")
oKP {
        Know("tea", "product")
        AddConversationQ("price").SetGoal(StzGoalQ().RequireEach("product", "price"))
        AskIn("price")
        ? @@(ReplyIn("price", "=12*3")[:admitted])
        #--> [ "=12*3" ]
        ? Query([ "tea", "price", "?o" ])[1]
        #--> =12*3
}
```

✓ exécuté : a affiché ce que ce bloc promet

Elle ne l'est pas. Le prix du thé est maintenant le texte `=12*3` ! Je l'ai découvert en l'exécutant, et le code est muet sur un registre de formule. Le dossier liste le travail de wise coding sur les registres 3 et 5 comme en attente, non acheminé.

Le cinquième registre, les exemples. Que fait la porte d'exemples offerts sous forme de paires ? L'agent qui les lirait figure sur la liste, et réservé :

HARO

```
oKX = new stzKnowledgeGraph("phones")
oKX {
        Know("contact1", "contact")
        AddConversationQ("fmt").SetGoal(StzGoalQ().RequireEach("contact", "number-pattern"))
        AskIn("fmt")
        aV = ReplyIn("fmt", [ [ "+216-21-298-374", "valid" ], [ "+33-6-12-34-56-78", "valid" ] ])
        ? @@(aV[:admitted])
        #--> [ ]
        ? @@(aV[:refused])
        #--> [ ]
        ? aV[:goalState]
        #--> pursuing
        ? len(ConversationQ("fmt").Checkpoints())
        #--> 0
}
try
        StzOwnAgent("induction")
catch
        ? cCatchError
done
#--> Softanza's own agent 'induction' is RESERVED, not wired yet -- refusing rather than
returning a stub (LAW 3). See StzOwnAgents().
```

✓ exécuté : a affiché ce que ce bloc promet

Ni admis ni refusés : `[ ]` et `[ ]`, le but toujours `pursuing`, et pas un seul point de contrôle ! Ce n'est pas le refus que décrit le protocole. C'est le silence. Je tiens le silence pour la pire réponse, comme une vue et non comme un résultat : le lecteur qui a offert les exemples n'est pas informé qu'ils n'ont pas été compris. J'ai trouvé une seconde forme de la même faute, avec les mêmes exemples tapés comme une seule chaîne, jointe par une virgule. L'exercice à la fin de ce chapitre vous la remet.

Ainsi trois des cinq registres sont câblés, l'un d'eux sous sa forme la plus simple, un séparateur. Deux ne le sont pas. Je préfère que vous le rencontriez ici plutôt que sur une page de moi qui aurait dit cinq.

Il reste le refus lui-même, et ce qu'il devient.

CE QUI ARRIVE À UN REFUS

Un refus attend une personne, avec son contexte. Une durée de vie peut être fixée pour qu'un refus périmé quitte la liste vivante sans être supprimé :

HARO

```
oKT = new stzKnowledgeGraph("ttl")
oKT {
        AddConversationQ("t").SetGoal(StzGoalQ().RequireOne("kitchen", "led-by"))
        AskIn("t")
        ReplyIn("t", 7)
        ? len(ConversationQ("t").Checkpoints())
        #--> 1
        ConversationQ("t").SetCheckpointTTL(1)
        AskIn("t")
        AskIn("t")
        ? len(ConversationQ("t").Checkpoints())
        #--> 0
        ? len(ConversationQ("t").AllCheckpoints())
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Rien n'est supprimé. L'expiration porte sur ce qui a encore besoin d'une personne, et la trace d'audit garde tout. Chaque appel ici est enchaîné depuis `ConversationQ`, parce qu'une copie conservée ne ferait silencieusement rien : le commentaire de la garde l'explique.

Une conversation est responsable d'un seul but, et elle s'achève en accomplissant le but ou en le révoquant au dossier. Une conversation révoquée ne demande rien et ne peut pas conclure :

HARO

```
oKG = new stzKnowledgeGraph("lifecycle")
oKG {
        Know("d1", "dish")
        Know("d2", "dish")
        AddConversationQ("s1").SetGoal(StzGoalQ().RequireEach("dish", "sauce"))
        AddConversationQ("s2").SetGoal(StzGoalQ().RequireEach("dish", "garnish"))
        ? ConversationQ("s1").GoalState()
        #--> pursuing
        AskIn("s1")
        ReplyIn("s1", "pesto")
        AskIn("s1")
        ReplyIn("s1", "pesto")
        ? ConversationQ("s1").GoalState()
        #--> fulfilled
        ? ConversationQ("s1").GoalWhy()
```

```
        #--> every required slot is filled in 'lifecycle'
        AskIn("s2")
        ConversationQ("s2").RevokeGoal("out of scope tonight")
        ? ConversationQ("s2").GoalState()
        #--> revoked
        ? ConversationQ("s2").GoalWhy()
        #--> out of scope tonight
        ? len(OpenConversations())
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

PROGRAMMATIQUE PAR DÉFAUT : LA CONVERSATION OÙ IL N'Y A PERSONNE

La loi du chapitre 17 dit que les conversations ont lieu d'abord dans le code. Voici la conversation sans personne et sans modèle. Un agent d'IA de la bibliothèque demande, une source de réponses (une simple fonction) répond, et chaque réponse passe par la même porte. Là où la source n'a pas de réponse, l'agent invente-t-il ? Exécutez-le :

HARO

```
oKW = new stzKnowledgeGraph("menu")
oKW.Know("margherita", "dish")
oKW.Know("tiramisu", "dish")
answers = [ [ "margherita", "tomato-sauce" ], [ "tiramisu", "mascarpone" ] ]
aR = StzOwnAgent("wise-coder").PursueGoal(StzGoalQ().RequireEach("dish", "contains"), oKW, func
subj, rel {
        for a in answers
                if a[1] = subj
                        return a[2]
                ok
        next
        return ""
})
? aR[:filled]
#--> 1
? aR[:asked]
#--> 2
? aR[:why]
#--> goal filled in 2 question(s), governed admission throughout
oKV = new stzKnowledgeGraph("desserts")
oKV.Know("panna-cotta", "dish")
aR2 = StzOwnAgent("wise-coder").PursueGoal(StzGoalQ().RequireEach("dish", "contains"), oKV, func
subj, rel { return "" })
? aR2[:filled]
#--> 0
```

```
? aR2[:why]
#--> stopped: no answer for 'panna-cotta' / 'contains' -- gap left for a human
```

✓ exécuté : a affiché ce que ce bloc promet

Il n'invente pas. Il s'arrête, dit `stopped: no answer for 'panna-cotta' / 'contains' -- gap left for a human`, et laisse l'écart à une personne. Des cinq agents propres de la bibliothèque, c'est celui qui est câblé. Les quatre autres sont réservés, et en demander un lève une erreur, comme l'a imprimé le bloc du cinquième registre : `RESERVED, not wired yet -- refusing rather than returning a stub`. Ce n'est jamais un bouchon.

EN ARABE, ET EN HAOUSSA

Les chapitres 2 et 11 ont montré la couche naturelle en haoussa. Voici ce que la conversation peut et ne peut pas faire avec les langues, dit en trois blocs, parce que les trois sont différents.

Premièrement, les données. Les noms, les valeurs et les listes en écriture arabe passent la barrière exactement comme les latins. Les mots autour d'eux, le cadre de la question, restent en anglais :

HARO

```
oKA = new stzKnowledgeGraph("مطعم")
oKA {
        Know("بيتزا", "dish")
        AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        ? AskIn("c")
        #--> What does 'بيتزا' have for 'contains'?  (why: every dish needs 'contains')
        aV = ReplyIn("c", [ "طماطم" ,"جبن" ])
        ? @@(aV[:admitted])
        #--> [ "طماطم" ,"جبن" ]
        ? aV[:goalState]
        #--> fulfilled
}
```

✓ exécuté : a affiché ce que ce bloc promet

Les valeurs arabes ont passé la barrière, et le but se lit `fulfilled`.

Deuxièmement, le candidat. Une phrase en haoussa peut produire une valeur dans la couche naturelle, et la valeur, envoyée comme réponse, passe par la même porte. Les deux étapes sont celles de l'appelant : la conversation n'appelle pas elle-même la couche naturelle. C'est le principe du chapitre en petit, puisque l'expression est libre et l'admission gouvernée. Le graphe replie la casse de la valeur, de sorte que le `JIBINI` imprimé est admis comme 'jibini' :

HARO

```
oKH = new stzKnowledgeGraph("menu")
oKH {
        Know("pizza", "dish")
        AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("dish", "contains"))
        AskIn("c")
}
cCandidate = NaturallyIn("ha", "Yi rubutu dauke 'jibini' Maida shi").Result()
? cCandidate
#--> JIBINI
aV = oKH.ReplyIn("c", cCandidate)
? aV[:narration]
#--> admitted: 'pizza' contains 'jibini'
```

✓ exécuté : a affiché ce que ce bloc promet

Troisièmement, la question dans la langue de l'apprenant. Le tuteur est un consommateur de la conversation : il utilise l'écart que la conversation nomme, et il choisit les mots pour le dire en quatre langues. Les phrases sont des gabarits. Le commentaire du code dit lui-même que les formulations en français, en arabe et en haoussa sont un brouillon en attente d'un relecteur natif. Lisez les trois ci-dessous comme des brouillons, et non comme un texte relu :

HARO

```
oEx = StzProgramQ("../../education/program").CourseQ("elementary-introduction").ExerciseQ("ex-01-
01")
oEx {
        ? GapText("finds", "en")
        #--> Where are the repeated items? Which line of your program asks for their positions?
(why: the mental model finds before it applies)
        ? GapText("finds", "fr")
        #--> Où sont les éléments répétés ? Quelle ligne de votre programme demande leurs
positions ? (pourquoi : le modèle mental trouve avant d'agir)
        ? GapText("finds", "ar")
```

```
        #--> أين العناصر المكررة؟ أيّ سطر في برنامجك يسأل عن مواضعها؟ (لماذا: النموذج الذهني يبحث قبل أن يطبّق)
        ? GapText("finds", "ha")
        #--> Ina abubuwan da aka maimaita? Wane layi a shirinka yake tambayar wurarensu? (dalili: tsarin tunani yana nemo kafin ya aiwatar)
}
```

✓ exécuté : a affiché ce que ce bloc promet

Ainsi la bibliothèque ne peut pas demander en arabe ou en haoussa par `AskIn`. Ce qu'elle peut faire, c'est tenir les données dans n'importe quelle écriture, prendre un candidat calculé dans n'importe quel paquet de langue, et laisser le tuteur dire l'écart en quatre langues. Une valeur qui contient un espace lève une erreur pour le moment (la section 7 donne la preuve). Et le séparateur ne connaît que la virgule et le mot anglais « and » : les connecteurs « et » en français, « da » en haoussa et « و » en arabe ne sont pas des connecteurs. Les trois phrases ci-dessus sont celles de la bibliothèque, imprimées telles quelles, et le code les appelle un brouillon en attente d'un relecteur natif. Je n'ai écrit moi-même aucune phrase en arabe ou en haoussa.

UN MODÈLE D'IA EST UNE FLUIDITÉ, JAMAIS L'ESPRIT

Sans modèle, que dit une conversation de sa propre fluidité ? Exécutez-le :

HARO

```
oKF2 = new stzKnowledgeGraph("fluency-space")
oKF2 {
        AddConversation("f")
        ? ConversationQ("f").Fluency()
        #--> plain
        ConversationQ("f").SetFluency(:neural)
        ? ConversationQ("f").Fluency()
        #--> neural
        ? ConversationQ("f").IsFluencyNeural()
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

`plain` : elle reste sur son plancher déterministe et le dit. Le mot « neural » peut être posé, et `Fluency()` le répète, mais `IsFluencyNeural()` répond `0` ! Le mot n'est revendiqué que lorsqu'un modèle est là.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé de deviner. La phrase du dossier est : deviner est remplacé par demander ; les impressions sont remplacées par la gouvernance. Un écart est une question. Une réponse est un candidat.

J'ai refusé une barrière différente pour chaque façon de répondre. Une formule n'est pas crue parce qu'elle est du code, et une structure n'est pas crue parce qu'elle est typée. La forme n'achète pas la confiance. Les cinq registres passent par une seule porte.

J'ai refusé un bouchon. Un agent réservé lève une erreur quand on le demande. Le document de conception appelle l'erreur levée la loi qui fonctionne, et dit qu'il ne faut pas la lire comme un défaut. J'ai refusé de faire semblant que le cinquième registre fonctionne.

J'ai refusé un agent d'IA qui comble un écart pour finir le travail. Le wise coder s'arrête, et laisse l'écart à une personne.

J'ai refusé qu'une conversation possède un monde. Elle a lieu à l'intérieur d'un espace de connaissances et n'en possède jamais un. Une copie de la connaissance tenue dans une session devient périmée.

J'ai refusé un point de contrôle supprimé. L'expiration retire un refus de la liste vivante. Le dossier le garde.

J'ai refusé que l'exemple ressemble à la réponse. La loi de clôture du dossier sur le cinquième registre est qu'un candidat admis réécrit son exemple comme promesse. Cette loi est conçue, et non construite.

5. LE TRIBUNAL

`wisecoding_narrated` prouve que l'écart est calculé à partir de l'espace, et que la question est menée par le système, nomme son sujet et dit pourquoi il demande. Elle prouve que « X and Y » et une structure de données sont admis. Elle prouve qu'une loi d'unicité refuse par la conversation et que le refus atteint un point de contrôle. Elle prouve enfin que conclure avec un écart ouvert est refusé. `wisecoding_rich_narrated` prouve le but suivi jusqu'à accompli ou révoqué, le cadre, et les options numérotées. Elle prouve le refus d'un numéro que personne n'a offert, la durée de vie d'un point de contrôle, et la fluidité qui n'est revendiquée qu'avec un modèle. `ownagentstack_narrated` prouve la liste de cinq, le wise coder câblé, l'agent réservé qui lève une erreur, et l'écart laissé à une personne. `tutor_gaps_narrated` prouve la question d'écart de l'exercice dans chaque langue, et qu'une étape sans question dans une langue est rouge, jamais une autre langue. Elle écrit des répertoires, aussi le livre la compte et ne l'exécute pas ; le bloc du tuteur ci-dessus ne fait que lire le cours.

Trois des gardes impriment leur propre total et se terminent en appelant `pf()`, qui imprime le temps et la version du runtime puis arrête l'exécution par une levée d'erreur. C'est leur façon de finir. `tutor_gaps_narrated` se termine par `Summary()`, qui imprime le total et un état, et ne lève pas d'erreur.

Ce que le terrain fait mieux, et où le dossier se distingue, vient ensuite.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`wisecoding_narrated` 13 assertions
base/test/conversation/wisecoding_narrated.ring

`wisecoding_rich_narrated` 52 assertions
base/test/conversation/wisecoding_rich_narrated.ring

`ownagentstack_narrated` 10 assertions
base/test/agentic/ownagentstack_narrated.ring

`tutor_gaps_narrated` 29 assertions
base/test/education/tutor_gaps_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en compréhension. Un cadre de dialogue, tel que le terrain le tient, lit le texte libre avec des modèles entraînés, parle sur de nombreux canaux et livre des connecteurs. Le dossier dit ce qui manque à Softanza, dans son propre compte : le mot « WhatsApp » n'apparaît dans aucun fichier de code ni aucun document de conception de la bibliothèque. La conversation demande un emplacement à la fois, sur un graphe que l'utilisateur doit d'abord avoir déclaré, et elle ne comprend pas une réponse ouverte.

Le terrain mène sur les exemples. La programmation par l'exemple est, telle que le terrain la tient, livrée dans les tableurs aujourd'hui. Softanza n'a aucune induction, et le dossier le dit : rien n'induit encore. Une recherche de « induce », « from examples » et « infer » dans le répertoire des expressions régulières ne trouve rien.

Le terrain est honnête sur les coûts, et le dossier aussi. Plusieurs candidats est le cas normal quand on demande une règle à partir d'exemples. Un motif induit de trois exemples les surapprend. Le remède que nomme le dossier est la question qui distingue deux candidats. Et l'index qui porterait les exemples est, selon les mots de l'étude, aussi propre que les étiquettes des méthodes, dans lesquelles l'étude compte environ quatre-vingt-cinq fautes d'orthographe.

Voici où je pense que le dossier se distingue. Chaque réponse, sous quelque forme que ce soit, est jugée par les lois du monde avant de compter, avec la raison. La session garde une transcription et un point de contrôle qu'une personne peut lire. La réponse du terrain est un prompt, et l'humain garde ce qui sort.

La limite vient ensuite, et elle nomme ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT	le but, l'écart et la question avec sa raison ; une réponse sous forme d'option, de structure de données ou de phrase découpée sur les virgules et sur le mot « and » ; la porte gouvernée avec ses refus ; le point de contrôle avec sa durée de vie ; le but accompli ou révoqué ; un agent d'IA câblé qui s'arrête là où il devrait inventer
CONÇU	plusieurs lectures d'une même réponse, énumérées pour que la personne choisisse ; les alternatives les plus proches nommées après un refus ; l'exemple réécrit comme promesse
ABSENT	le registre de la formule ; le registre de l'exemple ; une conversation dont la question est formulée en arabe, en haoussa ou en français ; une valeur qui contient un espace

Ce qui est construit : le but, l'écart, la question avec sa raison, le cadre ; le premier registre, le deuxième, et le quatrième sous la forme d'un séparateur ; la porte gouvernée avec ses refus. Et le point de contrôle et sa durée de vie ; le but accompli ou révoqué ; la transcription ; un agent câblé ; la phrase d'écart du tuteur en quatre langues.

Ce qui est conçu : la branche *plusieurs acceptables* telle que la conception l'écrit. Le protocole dit que lorsque plusieurs lectures d'une réponse sont acceptables, le système les énumère et la personne choisit. Ce qui est construit, ce sont deux choses différentes. Les valeurs déjà connues deviennent des options numérotées dans la question, avant la réponse. Et une réponse qui porte plusieurs valeurs voit chacune jugée à part, la première étant gardée sous une loi d'unicité. Aucune étape ne présente des lectures concurrentes d'une même réponse. Et la branche *aucune acceptable* telle qu'écrite : la raison est construite, et les alternatives acceptables les plus proches ne le sont pas. Le refus énonce une raison, et pour un numéro, combien d'options étaient sur la table. Et l'exemple réécrit comme promesse : la page du site *by example*, lue le 9 octobre 2026, l'appelle une règle de la conception, non construite. Les deux pages du site à ce sujet, *wise* et *by example*, disent « avec les alternatives les plus proches » et « la personne choisit ». Les deux pages promettent plus que le code. Le code câble trois registres sur cinq. Le dossier nomme la formule et les exemples comme en attente, non acheminés, et pour ce chapitre ils sont absents.

Ce qui est absent : le registre de la formule ; le registre de l'exemple, et l'agent d'induction derrière lui ; une conversation formulée en arabe, en haoussa ou en français ; et une valeur avec un espace. Lors de mon exécution du 8 octobre 2026, sur la version de la bibliothèque que ce livre exécute, `ReplyIn` a levé une erreur sur chaque valeur contenant un espace que j'ai essayée. L'erreur était une variable non initialisée. Les trois étaient « tomato sauce », « fromage et tomate », « jibini da tumatir ». Appelée directement, la porte a renvoyé correctement le refus. Aucun bloc de ce livre n'enregistre l'un ou l'autre résultat, de sorte que l'exécution du livre ne les montre pas. La conversation aurait dû refuser avec la raison et écrire un point de contrôle, et elle ne le fait pas. C'est une demande à la bibliothèque, et l'exercice ci-dessous n'emploie pas de valeur avec un espace.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Une réponse est un verdict que vous pouvez tester avec des promesses ordinaires : ce qui a été admis, ce qui a été refusé, et l'état du but.
L'analyste	Le but est le schéma d'un modèle complet. Les écarts sont la liste de ce qui manque. La transcription et les points de contrôle sont la preuve de la façon dont chaque fait est entré.
Le concepteur	La question revient sous forme de données : force, sujet, relation, options, raison. N'importe quel médium peut la rendre. La bibliothèque rend l'anglais, et le tuteur montre le motif pour quatre langues.
Le décideur	Une réponse est jugée avant de compter. Un refus arrive à une personne avec son contexte et une durée de vie. La liste est honnête : un agent sur cinq est câblé, et deux des cinq registres ne sont pas disponibles.
Le formateur	Le tuteur est du wise coding. Il nomme l'étape manquante comme une question et jamais comme la réponse, dans la langue de l'apprenant, et sa formulation dans trois de ces langues attend un relecteur natif.
L'agent	Vous pouvez piloter la boucle vous-même avec une source de réponses, et vous devez vous arrêter là où vous devriez inventer. Vous ne pouvez pas offrir des exemples et être compris, et on vous le dit par son nom quand vous demandez un agent réservé.

9. EXERCICE

Choisissez un petit monde à vous, une boutique, une classe, une clinique, avec quatre entités d'un même type et un but selon lequel chacune a besoin d'une relation. Avant d'exécuter quoi que ce soit, écrivez sous chacune des cinq réponses ci-dessous le verdict que vous attendez : admis, refusé avec une raison, ou ni l'un ni l'autre.

1. un mot simple ;
2. un numéro qui choisit une valeur que la première réponse a fait connaître ;
3. un numéro que personne n'a offert ;
4. deux valeurs à la fois, pour une relation que vous avez déclarée unique ;
5. une liste de deux paires d'exemples.

Puis exécutez-le. Une réponse admise épuise la question : appelez de nouveau `AskIn` avant la réponse suivante. Si une promesse se brise, ne changez pas la promesse : expliquez d'abord la sortie.

HARO

```
oWorld = new stzKnowledgeGraph("my-world")
oWorld.Know("item1", "thing")
oWorld.Know("item2", "thing")
oWorld.AddConversationQ("mine").SetGoal(StzGoalQ().RequireEach("thing", "colour"))
oWorld.AskIn("mine")
# your five replies here, each with the verdict you expect
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

La cinquième réponse produit du silence aujourd'hui, et c'est la leçon de l'exercice. Si la bibliothèque la transforme plus tard en refus, la réponse attendue change. Exécutez-le à la date et au commit indiqués au pied de la page.

Je vous laisse celui-ci. Prenez le bloc du cinquième registre de la section 3 et changez une seule chose : envoyez les deux numéros de téléphone comme une seule chaîne, jointe par une virgule, et non comme deux paires.

HARO

```
oKX = new stzKnowledgeGraph("phones")
oKX.Know("contact1", "contact")
oKX.AddConversationQ("fmt").SetGoal(StzGoalQ().RequireEach("contact", "number-pattern"))
oKX.AskIn("fmt")
aV = oKX.ReplyIn("fmt", "+216-21-298-374,+33-6-12-34-56-78")
? @@(aV[:admitted])
? aV[:goalState]
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le, puis lisez ce qui a été admis et ce que dit le but, et décidez si c'est ce que vous vouliez remettre à la porte en fait d'exemples. La section 4, dans son dernier refus, et la section 7, dans le registre des exemples absent, disent pourquoi j'y vois une faute. Quand vous saurez dire pourquoi, vous saurez ce que la porte a fait de vos exemples, et pourquoi ce n'est pas de l'induction.

C'est ce que j'appelle le wise coding. La plateforme pose la question à laquelle vous devez répondre, dans le registre que vous choisissez, et refuse de deviner la réponse à votre place : l'expression est libre, l'admission est gouvernée. Le chapitre 17 a énoncé la loi. Ce chapitre a fait tourner ses cinq registres, et montré où le cinquième reste encore muet.

Le chapitre suivant porte sur le modèle de langage de domaine. Là, la prochaine chose que dit le modèle est une unité vérifiable de votre monde, un fait que vos règles admettent ou refusent, et jamais un jeton.

OÙ SONT LES PREUVES

NARRATION	`stz-knowledge-programming-narration.md`
NARRATION	`stznatural-narration.md`
SUR LE SITE	wise
SUR LE SITE	byexample
DÉCISIONS DE L'ÉTUDE	12.25, 12.26, 12.27

PARTIE IV · CHAPITRE 19

Le modèle de langage de domaine

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 7 bloc(s) sur 7 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je forge un modèle de langage de domaine à partir d'une base de connaissances, et personne n'écrit son texte d'entraînement : chaque phrase est un fait enregistré. Il a deux échelons. Le premier répond à partir de ces faits et n'a aucun neurone. Le second, un petit modèle neuronal, est fluide dans la grammaire du domaine et se trompe sur les faits. Je montre la grammaire qui refuse une phrase malformée et le graphe qui refuse une phrase fausse. Je dis quelles pièces la bibliothèque contient et lesquelles le lecteur doit encore composer. Le principe est celui-ci : la prochaine chose que dit le modèle est une unité vérifiable de votre monde, un fait que vos règles admettent ou refusent, jamais un jeton.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Un modèle de langage prédit le jeton suivant. Un jeton est un fragment de texte sans type, sans rôle et sans contrainte : il peut être la fin d'un mot, un espace, un crochet. Tout ce qu'un modèle dit est construit en choisissant des jetons l'un après l'autre. Tout ce qui rend le résultat utile, que ce soit une phrase valide, qu'elle soit vraie, qu'elle respecte un schéma, est vérifié après coup ou imposé de l'extérieur.

Le terrain a fait fonctionner cela avec trois outils. On adapte un modèle d'IA général à un domaine en l'entraînant davantage sur les documents du domaine. On lui donne les documents pertinents au moment de la question, récupérés par similarité.

Et on contraint son décodage avec une grammaire, de sorte qu'il ne puisse pas produire une phrase malformée, comme l'a montré le chapitre 11. Tous trois gardent la même unité : le jeton.

La section suivante nomme la seule chose que je change : l'unité.

2. LE PREMIER PRINCIPE

La prochaine chose que dit le modèle est une unité vérifiable de votre monde, un fait que vos règles admettent ou refusent, jamais un jeton.

Dans la conception la barrière est la même pour tous les modèles de ce genre : les générations sont des propositions, et le graphe et ses règles décident de l'admission.

La section 3 montre la forme qu'il prend, et l'exécute.

3. LA FORME SOFTANZA

Je l'appelle un modèle de langage de domaine, et il a deux échelons. Le premier n'a aucun neurone et a été montré au chapitre 17 : il répond à partir de faits enregistrés, dit pourquoi, et refuse hors de son domaine. Le second est un petit modèle neuronal, facultatif, qui lui donne de la fluidité. Dans la conception, les deux échelons aboutissent à la même barrière ; ici, la barrière est l'`Admit` du dernier bloc de cette section, appelé à la main.

D'où vient le texte d'entraînement ? Construisez une petite base de connaissances, rendez-la, et lisez le corpus et le vocabulaire qu'elle donne.

HARO

```
oKB = new stzKnowledgeGraph("restaurant")
oKB {
        Know("margherita", "dish")
        Know("tiramisu", "dish")
        Know("chianti", "wine")
        KnowRelation("margherita", "contains", "tomato-sauce")
        KnowRelation("margherita", "contains", "mozzarella")
        KnowRelation("margherita", "pairs-with", "chianti")
}
oDLM = StzDlmQ(oKB)
oDLM {
        ? @@( GenerateCorpus() )
        #--> [ "margherita is a dish.", "tiramisu is a dish.", "chianti is a wine.", "margherita
```

```
contains tomato-sauce.", "margherita contains mozzarella.", "margherita pairs-with chianti." ]
        ? @@( Tokenizer() )
        #--> [ "<unk>", "margherita", "tiramisu", "chianti", "dish", "wine", "contains", "pairs-
with", "is", "a", "tomato-sauce", "mozzarella" ]
        ? @@( Tokenize("margherita contains something-new") )
        #--> [ 2, 7, 1 ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Personne n'a écrit ce corpus : chacune de ses six phrases est un fait de la base de connaissances, rendu. Le vocabulaire est fermé, chaque mot que le domaine contient avec une première entrée pour l'inconnu. Douze entrées, et `something-new`, un mot hors du domaine, est envoyé sur la première : `[ 2, 7, 1 ]`. « Correct par construction » vaut ici dans le seul sens que le code lui donne : chaque phrase est un fait enregistré. Cela ne vaut pas encore dans le sens que la conception visait ; la section 7 dit où.

La complétion est contrainte au domaine. Qu'offre le modèle après une entité qui porte des faits, après une relation, après un fragment, et après un mot que le domaine ne contient pas ? Exécutez-le :

HARO

```
oDLM {
        ? @@( Complete("margherita") )
        #--> [ "contains", "pairs-with" ]
        ? @@( Complete("contains") )
        #--> [ "tomato-sauce", "mozzarella" ]
        ? @@( Complete("tira") )
        #--> [ "tiramisu" ]
        ? @@( Complete("pizza") )
        #--> [ ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Après une entité qui porte des faits, ses relations : `[ "contains", "pairs-with" ]`. Après une relation, seulement ses objets : `[ "tomato-sauce", "mozzarella" ]`. Un fragment se complète en entités, et un mot que le domaine ne contient pas se complète en rien : `[ ]`. La garde qui le prouve l'appelle « the grammar spirit ». C'est

une consultation dans les faits, non un décodage sous une grammaire compilée. Et elle voit un mot, pas la phrase jusque-là : `Complete("contains")` renvoie les objets de chaque fait `contains`, quel que soit son sujet.

La grammaire elle-même peut être dérivée de la base de connaissances, et ici le lecteur de ce livre doit écrire la dérivation. Il n'y a dans la bibliothèque aucun appel qui transforme une base de connaissances en grammaire, et la conception en fait la tâche du premier échelon. Mais les pièces se composent. Le vocabulaire du domaine devient les choix d'un schéma du chapitre 11, le schéma se compile en grammaire, et la grammaire n'a aucun chemin pour un mot hors du domaine. Les trois règles qu'elle imprime en dernier, pour le texte, le nombre et le booléen, sont celles que le compilateur émet toujours, et aucun champ ici ne les emploie :

HARO

```
aLex = oDLM.Lexicon()
acObjects = []
for aFact in oKB.Facts()
        if aFact[2] != "is-a" and find(acObjects, aFact[3]) = 0
                acObjects + aFact[3]
        ok
next
oS = StzOutputSchemaQ([
        [ :field = "subject",  :type = :oneof, :choices = aLex[:entities] ],
        [ :field = "relation", :type = :oneof, :choices = aLex[:relations] ],
        [ :field = "object",   :type = :oneof, :choices = acObjects ]
])
? oS.ToGBNF()
#--> root ::= subject-line relation-line object-line
#    subject-line ::= "subject: " ("margherita" | "tiramisu" | "chianti") "\n"
#    relation-line ::= "relation: " ("contains" | "pairs-with") "\n"
#    object-line ::= "object: " ("tomato-sauce" | "mozzarella" | "chianti") "\n"
#    text ::= [^\n]+
#    number ::= "-"? [0-9]+ ("." [0-9]+)?
#    boolean ::= ("yes" | "no" | "true" | "false")
cGood = "subject: margherita" + nl + "relation: contains" + nl + "object: mozzarella" + nl
cOut  = "subject: pizza" + nl + "relation: contains" + nl + "object: mozzarella" + nl
cLie  = "subject: tiramisu" + nl + "relation: contains" + nl + "object: mozzarella" + nl
? oS.Accepts(cGood)
#--> 1
? oS.Accepts(cOut)
#--> 0
? oS.Accepts(cLie)
#--> 1
? oDLM.Ask("tiramisu contains what?")
#--> I hold no 'contains' knowledge about 'tiramisu'.
```

✓ exécuté : a affiché ce que ce bloc promet

Regardez ensemble les quatre dernières lignes. La grammaire accepte le mensonge ! Il a la bonne forme, et chaque mot est dans le domaine. Interrogé sur ce que contient un tiramisu, le modèle répond : `I hold no 'contains' knowledge about 'tiramisu'.` Une grammaire est une barrière pour la forme ; la vérité appartient au graphe.

Un modèle de langage de domaine est un fichier qui voyage. Il est sauvegardé en texte simple, lisible et comparable par diff, sans binaire et sans poids. Un second modèle qui n'a jamais lu la base de connaissances peut-il répondre à partir de lui, et exécuter ses propres tests ? Sauvegardez-le, relisez-le, et chargez-le :

HARO

```
oDLM {
        AddGolden("what is margherita", "Margherita is a dish.")
        AddGolden("what is the weather today?",
                "That is outside the 'restaurant' domain (I know 3 entities and 2 relations).")
        cFile = Save("probe_dlm")
}
? cFile
#--> probe_dlm.zdlm
? read(cFile)
#--> dlm "restaurant"
#    entities
#        margherita | dish
#        tiramisu | dish
#        chianti | wine
#    relations
#        contains
#        pairs-with
#    facts
#        margherita | contains | tomato-sauce
#        margherita | contains | mozzarella
#        margherita | pairs-with | chianti
#    laws
#    goldens
#        what is margherita | Margherita is a dish.
#        what is the weather today? | That is outside the 'restaurant' domain (I know 3 entities
and 2 relations).
oShipped = StzDlmQ("")
oShipped {
        LoadFrom(cFile)
        ? Ask("what is tiramisu")
        #--> Tiramisu is a dish.
```

```
        ? @@( RunGoldens() )
        #--> [ [ "total", 2 ], [ "passed", 2 ], [ "failed", [ ] ] ]
}
remove(cFile)
? fexists(cFile)
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le second modèle répond : `Tiramisu is a dish.` Le refus voyage avec le lot comme test golden, et le modèle qui n'a jamais lu le graphe le passe aussi ! La section des lois est vide parce qu'aucune loi n'a été déclarée avant que le modèle soit forgé ; déclarez qu'une relation est unique, forgez de nouveau, et la ligne apparaît.

Maintenant le second échelon. C'est un petit modèle neuronal entraîné sur le corpus que vous avez vu, sans enseignant, et il peut être exporté comme fichier de modèle. Quelle est sa taille, et apprend-il ?

HARO

```
oDLM {
        TrainNeuralRung(500)
        ? NeuralWhy()
        #--> neural bigram rung: 15 pairs over a 12-word vocab; trained 500 epoch(s), lr 0.50:
loss 2.57 -> 0.70
        ? NextToken("is")
        #--> a
        ? NextTokenConfidence("is")
        #--> 1.00
        cGguf = ExportNeuralGguf("probe_bigram")
}
? cGguf
#--> probe_bigram.gguf
? StzEngineNeuralGgufInspect(cGguf)
#--> 2
remove(cGguf)
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Il est petit : une seule couche qui prédit le mot suivant à partir du précédent, `15 pairs over a 12-word vocab`. Et il apprend : `loss 2.57 -> 0.70`. La bibliothèque inspecte le fichier exporté et imprime 2, les deux tenseurs qu'il contient, un poids et un biais. Je ne prétends pas qu'un autre runtime le charge comme générateur de texte. Il s'entraîne en une fraction de seconde, hors ligne, et donne les mêmes nombres à chaque exécution.

Que dit un modèle neuronal comme celui-ci ? Il est fluide dans la grammaire du domaine, et il se trompe sur les faits. C'est la raison de la barrière :

HARO

```
aCorpus = oDLM.GenerateCorpus()
for cSeed in [ "tiramisu", "chianti", "margherita" ]
        cSaid = oDLM.NeuralGenerate(cSeed, 3)
        ? cSaid + "  ->  " + ( find(aCorpus, cSaid + ".") > 0 )
next
#--> tiramisu is a wine  ->  0
#    chianti is a wine  ->  1
#    margherita contains mozzarella is  ->  0
```

✓ exécuté : a affiché ce que ce bloc promet

« Tiramisu is a wine » est faux, parce que le tiramisu est un plat. « Chianti is a wine » est vrai. Et « margherita contains mozzarella is » est une phrase qui n'en finit pas, parce que le bigramme n'a aucun mot qui termine une phrase. Le test contre le corpus est une barrière grossière, et il refuserait un fait vrai que personne n'a enregistré. La vraie barrière est le graphe et ses lois. Que répond le graphe quand la fausse phrase lui est proposée comme un fait ? Déclarez la loi, puis proposez :

HARO

```
oKB {
        ConstrainRelation("is-a", :Unique)
        aVerdict = Admit("tiramisu", "is-a", "wine", [ :source = "rung 2" ])
        ? aVerdict[:admitted]
        #--> 0
        ? aVerdict[:why]
        #--> 'tiramisu' already bears 'is-a' (to 'dish'); the relation is :Unique
        ? @@( Contradictions() )
        #--> [ [ [ "subject", "tiramisu" ], [ "relation", "is-a" ], [ "existing", "dish" ], [
"attempted", "wine" ], [ "source", "rung 2" ] ] ]
```

```
        ? Admit("pizza", "is-a", "dish", [ :source = "rung 2" ])[:why]
        #--> admitted: 'pizza' is-a 'dish'
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le graphe a refusé la fausse proposition ! Il dit pourquoi : `'tiramisu' already bears 'is-a' (to 'dish'); the relation is :Unique`. Le refus est gardé, avec sa source : le second échelon du modèle. Un fait nouveau est admis. C'est tout le principe en six appels : une proposition, une loi, un refus avec son pourquoi, et le refus au dossier.

La conception décrit l'unité de prédiction en trois niveaux. Un jeton est un fragment sans type. Un plongement est un vecteur sans rôle. Et un chunk est un nœud typé d'un graphe, avec une étiquette, une charge utile et des arêtes : une déclaration, une relation, une contrainte. Un chunk est validé en cinq étapes avant d'entrer dans le graphe : type, motif, intégrité du graphe, conformité, et contribution constructive. Je le raconte comme conception, parce qu'il n'y a pas de chunk dans la bibliothèque. Ce qui existe, ce sont des pièces : le tribunal du chapitre 11 vérifie le type et la forme d'un texte ; et l'admission ci-dessus impose une loi, ce qui est la conformité. Rien ne les enchaîne.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Le 13 juillet 2026, j'ai tranché sur ce point. Le concept d'un modèle de langage de domaine avait été conçu pour l'un des produits du domaine de projets. Je l'ai donné à la fondation ouverte, pour que tout projet Softanza qui fournit une base de connaissances puisse livrer son propre modèle gratuitement aux utilisateurs de son domaine. Le format est `.zdlm`. Le premier échelon est dans chaque modèle ; le second est une fluidité facultative. Le dossier tient la décision qu'un modèle se livre gratuitement, et aucun texte de licence. Je dis « gratuit » comme le mot de cette décision : un fichier simple, dans la fondation ouverte, sans redevance de licence ni compteur.

J'ai refusé le jeton comme unité. L'unité est un chunk du monde, qu'un graphe peut admettre ou refuser.

J'ai refusé un enseignant que je ne peux pas voir. La conception antérieure de l'idée employait un grand modèle d'IA commercial pour écrire son corpus d'entraînement. J'ai posé une loi contre cela, et le corpus de ce modèle est rendu à partir des faits propres du domaine : une loi de la bibliothèque dit qu'une capacité n'exige jamais rien de payant ni de distant.

J'ai refusé le neuronal d'abord. La découverte dont je pars est que le modèle canonique n'est pas neuronal. Une revue du produit antérieur, consignée dans la conception que le domaine de projets en fait, a archivé un transformeur de un à trois milliards de paramètres au profit d'une prédiction à base de règles. Son motif était que l'hallucination est impossible quand le modèle ne peut pas générer de texte presque valide. Le premier échelon a donc zéro entraînement et zéro coût, et l'échelon neuronal vient en second.

J'ai refusé que la forme tienne lieu de vérité. La grammaire dit que la structure tue la malformation, pas la fausseté. La vérité appartient au graphe, à ses lois, et à une personne.

La section 5 dit ce que le tribunal en prouve.

5. LE TRIBUNAL

`dlm_narrated` prouve le modèle forgé à partir d'un graphe, ses réponses avec leur pourquoi, son refus hors du domaine et la complétion. Il prouve aussi le lot en texte simple avec ses goldens, le tokeniseur fermé et le second échelon avec son fichier de modèle. Une de ses assertions sur le second échelon vérifie seulement que la phrase générée commence par un préfixe fixe, de sorte qu'elle passerait aussi sur la fausse phrase ci-dessus. `structuredoutput_narrated`, `schema_grammar_narrated` et `constrained_decoding_narrated` prouvent le tribunal du chapitre 11, le compilateur de grammaire et l'échantillonneur. La dernière a deux scènes qui exigent un fichier de modèle génératif, que l'instantané ne détient pas ; le dossier compte 50 assertions dans cette garde. `knowledge_integration_narrated` prouve la dérivation et le refus par une loi. L'admission que je montre dans le dernier bloc n'est appelée par son nom dans aucune garde. On l'atteint par la conversation, et par la forme stricte d'ajout d'un fait, où un fait ajouté sans son admission est refusé. Cela, la garde l'affirme bien.

La section 6 demande où le terrain mène, et où ce modèle se situe.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`dlm_narrated` 16 assertions base/test/learning/dlm_narrated.ring

`structuredoutput_narrated` 73 assertions
base/test/neural/structuredoutput_narrated.ring

`schema_grammar_narrated` 28 assertions
base/test/neural/schema_grammar_narrated.ring

`knowledge_integration_narrated` 19 assertions
base/test/natural/knowledge_integration_narrated.ring

`constrained_decoding_narrated` 50 assertions
base/test/neural/constrained_decoding_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en échelle. Entraîner et servir des modèles d'IA de toute taille est une pratique mûre. Ce second échelon est une seule couche sur douze mots, et les étapes qui le feraient devenir plus, un corpus validé, la déduplication, un tokeniseur entraîné, un réseau plus grand, la quantification, ne sont pas faites. Je ne rivalise pas avec les grands modèles d'IA.

Le terrain mène en fluidité, ce qui est le but d'un modèle d'IA. Un premier échelon qui répond à partir des faits dira « I hold no 'contains' knowledge about 'tiramisu'. » là où un modèle d'IA général écrirait quelque chose de plausible. Et pour un client la chose plausible est souvent plus utile. La réponse propre du dossier est le second échelon, et son état présent est celui que donne la section 7.

Là où je pense que ceci se distingue, c'est dans ce qu'est la sortie. La réponse d'un modèle d'IA général est du texte, et chaque vérification sur elle se fait après coup. Ici chaque réponse du premier échelon porte son pourquoi et une certitude de un. Chaque proposition du second peut être soumise à une barrière, et la barrière peut la refuser avec une raison. Un modèle qui ne peut dire que ce que le domaine contient, et une barrière qui se souvient de ce qu'elle a refusé : voilà ce qu'une banque, un ministère ou une école peut signer. Je tiens cela pour une vue, non pour un résultat.

La section 7 dit ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT	le premier échelon du modèle de domaine sans neurones ; ses faits rendus comme un corpus ; un vocabulaire fermé ; la complétion par consultation ; un lot en texte simple qui voyage avec ses tests ; un petit second échelon, un bigramme, exporté vers un fichier de modèle ; l'admission d'un fait par une loi
CONÇU	une grammaire dérivée d'une base de connaissances par un seul appel ; l'unité typée de prédiction validée en cinq étapes ; un second échelon dont le corpus est vérifié avant qu'il s'entraîne ; la fonderie qui l'agrandirait
ABSENT	un pipeline qui enchaîne les cinq étapes sur n'importe quelle unité ; un mot de fin de phrase pour le second échelon ; une complétion qui voit la phrase jusque-là

Ce qui est construit : le premier échelon et ses réponses ; les faits rendus comme corpus, avec un vocabulaire fermé ; la complétion par consultation ; et le lot qui voyage avec ses tests. Construit aussi : un second échelon qui s'entraîne en une fraction de seconde et exporte un petit fichier de modèle, et l'admission d'un fait par une loi.

Ce qui est conçu tient en quatre points.

- La grammaire dérivée d'une base de connaissances par un seul appel. La composition de ce chapitre est celle du lecteur, pas celle de la bibliothèque.

- Le corpus du second échelon validé par la grammaire du premier avant d'être admis. La conception le dit ; le code rend les faits directement, sans une telle vérification. Ainsi « correct par construction » vaut pour le texte d'entraînement, et non pour ce que dit le modèle.
- La validation en cinq étapes d'un chunk. Aucune n'existe comme pipeline.
- Les étapes de la fonderie au-delà du bigramme.

Ce qui est absent : un mot de fin de phrase pour le second échelon, ce qui explique qu'une de ses phrases n'en finisse pas. Absente aussi, une complétion qui voit la phrase jusque-là ; c'est pourquoi une complétion après une relation énumère les objets de chaque fait de cette relation. Le premier échelon répond aussi à une question reformulée avec le type, comme l'a montré le chapitre 17, là où la loi de cette partie dit qu'il devrait refuser.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Un modèle de langage de domaine est un fichier que vous pouvez lire, comparer par diff et tester : ses faits, ses lois et ses goldens.
L'analyste	Un fait que vous ajoutez fait que le modèle en sait plus, et une loi que vous ajoutez fait refuser davantage le graphe, sans entraînement ; les réponses propres du modèle ne la lisent pas encore.
Le concepteur	Le refus du modèle fait partie de son contrat : le test golden dit ce qu'il doit refuser de répondre.
Le décideur	Le modèle ne peut dire que ce que le domaine contient, et la barrière garde la trace de ce qu'elle a refusé et pourquoi.
Le formateur	Le premier échelon est un modèle qu'un apprenant peut lire en entier. Le second est assez petit pour être entraîné devant une classe.
L'agent	Ce que vous proposez est un fait que le graphe peut admettre ou refuser, avec une raison que vous pouvez lire.

9. EXERCICE

Forgez un domaine à vous avec trois entités et deux relations. Avant d'exécuter quoi que ce soit, écrivez ce que vous attendez de trois appels. Ce sont la complétion d'une entité, la complétion d'une relation, et la réponse à une question sur une chose que le domaine ne contient pas. Puis déclarez qu'une relation est unique et proposez, sous forme de fait, quelque chose qui la contredit.

Exécutez-le. Puis écrivez une phrase de la bonne forme et de la mauvaise vérité, et dites lequel de la grammaire et du graphe la refuse, et lequel ne la refuse pas.

Je vous laisse celui-ci. La section 3 dit que la complétion voit un mot et non la phrase jusque-là, et la section 7 la range parmi ce qui est absent. Dans votre domaine, donnez à deux entités la même relation avec des objets différents, demandez la complétion de cette relation, et lisez la liste contre la phrase que vous aviez en tête. La bibliothèque ne le répare pas, et je ne l'ai pas réparé pour vous. Puis composez, avec ce que la section 3 montre (les faits d'une base de connaissances, lus un par un), une complétion qui voit le sujet en plus de la relation.

Quand vous l'aurez résolu, vous saurez pourquoi une complétion qui voit un mot énumère les objets de chaque fait de cette relation.

Le chapitre 20 passe de ce qu'un modèle d'IA dit à ce qu'un agent d'IA fait. Son titre est *Des agents qui ne peuvent pas vous nuire*, et son principe est celui-ci : ne construisez pas des agents sûrs ; construisez un monde sûr, et lâchez-y des agents ordinaires.

OÙ SONT LES PREUVES

NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stz-knowledge-programming-narration.md`
SUR LE SITE	languages
SUR LE SITE	agentic
DÉCISIONS DE L'ÉTUDE	12.19, 12.29, 12.30

PARTIE IV · CHAPITRE 20

Des agents qui ne peuvent pas vous nuire

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends la moitié de l'action de ce que le chapitre 17 a dit de l'esprit entier : ce qui se passe quand la proposition rencontre le monde. Je montre quatre étapes en code : l'agent d'IA propose, l'atelier répète, le tribunal juge, et seul un acteur gouverné valide. Viennent ensuite les cas où une étape ne tient pas. Je dis ce que j'ai refusé : les agents sûrs, un contrôle de permission comme frontière, un second atelier pour les machines, une barrière facultative. Et je dis où cela s'arrête : un plan sans aucun exécutant est permis, et la boucle vérifie qu'une déclaration de couverture existe, non qu'elle est vraie. Le principe est celui de la doctrine : ne construisez pas des agents sûrs ; construisez un monde sûr, et lâchez-y des agents ordinaires.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Sur le terrain, la sûreté est une propriété de l'agent d'IA. Le document de doctrine de la bibliothèque énumère ce que le terrain offre : l'alignement, les garde-fous, les invites de confirmation, une case à cocher qui dit qu'un humain est dans la boucle. Les agents à outils, et les agents autonomes qui en descendent, laissent les appels d'outils du modèle toucher la réalité immédiatement, avec une invite de confirmation. Les agents d'usage de l'ordinateur tournent dans une machine

virtuelle en bac à sable, qui contient un processus et qui, dans les mots de la doctrine, ne distingue pas un déplacement de fichier d'un brassage d'octets aléatoires.

Le meilleur conseil du terrain, que, selon la narration, tout guide sérieux de 2026 donne (elle paraphrase une feuille de route), est de séparer la planification de l'exécution et de placer entre les deux une approbation humaine. C'est un bon conseil, et c'est un conseil : un patron que vous implémentez, et que vous pouvez oublier. Les listes de menaces du terrain, dont la liste de risques de l'OWASP pour les applications agentiques, sont des lentilles, et le dossier s'en sert ainsi.

Donc la posture du terrain, en une ligne, est que l'agent est un modèle d'IA dans une boucle, et que la sûreté est de la vigilance autour de lui. C'est de la connaissance générale, et non une étude du dossier, sauf là où je nomme le document. Le chapitre 17 l'a dit de l'esprit entier. Ce chapitre prend la moitié de l'action : ce qui se passe quand la proposition rencontre le monde.

2. LE PREMIER PRINCIPE

Ne construisez pas des agents sûrs. Construisez un monde sûr, et lâchez-y des agents ordinaires.

La phrase est dans le document de doctrine de la bibliothèque : « We will not build safe agents. We will build a safe world, and let ordinary agents loose inside it. » Le document en donne la moitié structurelle en une phrase de plus. Le seul artefact qui quitte jamais son monde est un plan de mise à jour, et les plans sont exécutés par quelqu'un d'autre. Le document ne porte ni date ni ligne d'auteur dans son en-tête, et git l'enregistre pour la première fois dans un commit du 9 juillet 2026. Le chapitre tient la phrase pour celle du document, datée par ce premier commit, et ne donne pas de date d'écriture.

3. LA FORME SOFTANZA

Quatre étapes, dans la forme que l'étude du dossier leur donne : l'agent d'IA propose, l'atelier répète, le tribunal juge, et seul un acteur gouverné valide. Je montrerai chacune en code, puis les cas où une étape ne tient pas.

Cet agent peut-il exister ? Avant qu'un agent ne fasse un seul tick, la gouvernance pose la question, et elle la pose quand la réponse est gratuite. Un agent qui ne déclare aucune couverture, une phrase disant ce qu'il couvre, est refusé. Un agent qui ne déclare aucune classe de réversibilité est refusé aussi : réversible, compensable ou irréversible. Les deux refus sont dans les phrases du moteur lui-même, et la porte de l'ancien langage hôte les répète. Que dit-elle à un agent qui ne déclare aucune couverture, et à un agent qui ne déclare aucune classe de réversibilité ? Exécutez-le :

HARO

```
oGov = new stzGovernance("shop")
oGov {
        DeclareReversibility("restocker", :Compensable)
        ? ReversibilityOf("restocker")
        #--> compensable
        ? MayRegister("mystery", "", "reversible")
        #--> 0
        ? Why()
        #--> AGENTLOOP-R4: agent 'mystery' declares no coverage statement. Registration refuses
it. Say what this agent covers, in one sentence, before the loop will run it. Law 18: nothing
schedules what nobody has stated the reach of.
        ? MayRegister("mystery", "reorders stock", "")
        #--> 0
        ? Why()
        #--> AGENTLOOP-R5: agent 'mystery' declares no reversibility class. Registration refuses
it. Declare one of: reversible | compensable | irreversible. Law 18: an agent whose reversal
nobody stated cannot be scheduled by something that cannot undo it.
        ? MayRegister("mystery", "reorders stock", :Compensable)
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le premier refus se termine sur sa loi : rien ne planifie ce dont personne n'a énoncé la portée. Le numéro qu'il porte appartient à la liste des lois propre à la bibliothèque, que ce livre n'imprime pas. Avec les deux déclarées, le troisième appel répond `1`.

La réversibilité est le sixième contrat de la gouvernance, à côté du risque, de l'autorité, de la validation, de la mise hors service et de la filiation. Elle est orthogonale au risque : un acte de faible risque qui ne peut pas être défait mérite plus de cérémonie qu'un acte de grand risque qui peut l'être.

Plus un acte est difficile à défaire, plus le code qui l'accomplit doit être digne de confiance. Quelles postures peuvent accomplir quelles classes d'acte ? Voici la table entière, imprimée par une boucle propre au livre :

HARO

```
aPostures = [ "trusted", "external", "sandboxed" ]
aClasses = [ "reversible", "compensable", "irreversible" ]
for cP in aPostures
        cRow = cP + ":"
        for cC in aClasses
                if StzPostureReversibilityRefusal(cP, cC) = ""
                        cRow += " yes"
                else
                        cRow += " no"
                ok
        next
        ? cRow
next
#--> trusted: yes yes yes
#--> external: yes yes no
#--> sandboxed: yes no no
? StzPostureReversibilityRefusal("sandboxed", "irreversible")
#--> a 'sandboxed' posture does not cover 'irreversible' work -- the harder an act is to undo,
the more trusted the code performing it must be (reversibility x posture, ruling 3.2)
```

✓ exécuté : a affiché ce que ce bloc promet

Lisez les lignes de haut en bas : le code de confiance peut accomplir les trois classes d'acte, le code externe les deux premières, le code en bac à sable seulement la réversible. La dernière ligne demande le refus et imprime sa phrase.

L'agent s'exécute, croit avoir agi, et le disque ne bouge pas. Un agent d'IA est un fichier et quelques fonctions. Sa fonction écrit une liste de prix. On lui donne un atelier, et la fonction atteint l'atelier comme un objet ambiant pendant un cycle, sans changement dans sa signature. La compétence vérifie son propre travail et le fait est appris. Après un cycle, le vrai fichier existe-t-il ? Interrogez le disque, puis l'atelier :

HARO

```
StzAllowAgentFunction("ScribblePrices", "trusted")
cFx = StzReplace(WorkingDirectory(), "\", "/") + "/_fixture"
```

```
StzDirCreatePath(cFx)
$cPrices = cFx + "/prices.csv"
cPia = "pia: 2
name: scribbler
kind: pi
coverage: writes a price list and records that it did
reversibility: compensable
schedule:
  timer: 20
skills:
  - name: scribble
    when: always
    does: ring:ScribblePrices
    verify: fact prices were written
    posture: trusted
"
oAg = StzAgentDeclarationQ(cPia).ToAgent()
oAg {
        GiveWorkbench()
        ? Cycle()
        #--> 1
        ? MemoryQ().Fact("prices", "were", "written")
        #--> 1
        ? StzFileExists($cPrices)
        #--> 0
        ? WorkbenchQ().Exists($cPrices)
        #--> 1
        ? WorkbenchQ().ContentOf($cPrices)
        #--> espresso,2.40
        #--> flat-white,3.10
}

func ScribblePrices(poMemory)
        StzActiveWorkbenchQ().WriteFile($cPrices, "espresso,2.40" + char(10) + "flat-white,3.10")
        poMemory.Learn("prices", "were", "written")
```

✓ exécuté : a affiché ce que ce bloc promet

Le fichier vit dans le jumeau. L'agent croit avoir agi, et le disque n'a pas bougé !

Deux lignes décident qui peut agir. `posture: trusted` est dans le fichier, parce qu'une clause `ring:` doit porter une posture. `StzAllowAgentFunction` est une ligne du code de la plateforme, et non du fichier de l'agent, et elle décide quelles fonctions un fichier peut appeler.

Le seul export est un plan, et il se narre lui-même. Le tick n'exporte vers la réalité qu'un plan, et rien d'autre. Le plan dit ce qu'il ferait avant que quoi que ce soit ne soit tenté. Un modèle de langage désigné comme son exécutant est refusé avant

toute tentative : `actor 'eager-llm' cannot commit -- it lacks the 'effectful' capability (required by operation 1)`. Un acteur nommé sous une portée qui ne couvre pas le chemin ne valide rien : le disque n'a toujours pas bougé. Le même acteur sous une portée qui couvre le chemin valide une fois, et la piste d'audit se lit `[ [ 1, "committed", "write_file", "shop-committer" ] ]`. Qui peut valider, et quand le disque bouge-t-il enfin ? Dans ce bloc, la réalité change ici, et seulement ici :

HARO

```
oPlan = oAg.GenerateUpdatePlan()
? StzReplace(oPlan.Narration(), cFx, "<fixture>")
#--> Update plan (1 of 1 operations to commit):
#-->   * 1. write 29 bytes to '<fixture>/prices.csv'
oPlan.SetExecutor(LLMActor("eager-llm"))
? oPlan.MayCommit()[2]
#--> actor 'eager-llm' cannot commit -- it lacks the 'effectful' capability (required by
operation 1)
oScope = new stzCommitScope()
oScope.AllowUnder("/somewhere/else")
oPlan2 = oAg.GenerateUpdatePlan()
oPlan2.SetExecutor(PIActor("shop-committer"))
oPlan2.SetScope(oScope)
oPlan2.Execute()
? StzFileExists($cPrices)
#--> 0
oScope3 = new stzCommitScope()
oScope3.AllowUnder(cFx)
oPlan3 = oAg.GenerateUpdatePlan()
oPlan3.SetExecutor(PIActor("shop-committer"))
oPlan3.SetScope(oScope3)
oPlan3.Execute()
? StzFileExists($cPrices)
#--> 1
? @@(oPlan3.AuditTrail())
#--> [ [ 1, "committed", "write_file", "shop-committer" ] ]
StzDirDeleteAll(cFx)
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

J'ai remplacé le chemin absolu dans la narration pour que la sortie soit stable. Un plan inclut le chemin du fichier qu'il écrirait.

Le tribunal au chargement. La table ci-dessus est-elle appliquée quand le fichier est chargé, avant le premier tick ? Prenez le même fichier, rendez l'agent irréversible, et mettez son code en bac à sable :

HARO

```
cIrrev = StzReplace(StzReplace(cPia, "reversibility: compensable", "reversibility:
irreversible"), "posture: trusted", "posture: sandboxed")
oD = StzAgentDeclarationQ(cIrrev)
? oD.IsValid()
#--> 0
? oD.CiteFindings()
#--> [pia-posture-reversibility @ skills.scribble.posture] a 'sandboxed' posture does not cover
'irreversible' work -- the harder an act is to undo, the more trusted the code performing it must
be (reversibility x posture, ruling 3.2)
```

✓ exécuté : a affiché ce que ce bloc promet

Le tribunal répond par la phrase même de la table : `a 'sandboxed' posture does not cover 'irreversible' work`. La même règle, au chargement : le fichier est refusé avant son premier tick !

Une proposition qui tourne mal. Un rangement qui supprime chaque fichier d'un répertoire, mené ici à la main à travers le jumeau comme le ferait un agent, propose trente suppressions. Le plan liste trente opérations et en signale trente comme des risques. Un modèle de langage qui demande à valider est refusé. Et quand il appelle `Execute()` malgré tout, combien de fichiers reste-t-il ? Comptez-les :

HARO

```
cDir = StzReplace(WorkingDirectory(), "\", "/") + "/_course"
StzDirCreatePath(cDir)
for i = 1 to 30
        StzFileWrite(cDir + "/lesson" + i + ".txt", "lesson " + i)
next
oBench = new stzVirtualFileSystem()
oBench.SetActor("tidy-bot")
for i = 1 to 30
        oBench.MirrorFile(cDir + "/lesson" + i + ".txt")
        oBench.DeleteFile(cDir + "/lesson" + i + ".txt")
next
oBig = oBench.GenerateUpdatePlan()
```

```
? oBig.NumberOfOperations()
#--> 30
? len(oBig.Risks())
#--> 30
oBig.SetExecutor(LLMActor("helper-llm"))
? oBig.MayCommit()[2]
#--> actor 'helper-llm' cannot commit -- it lacks the 'effectful' capability (required by
operation 1)
oBig.Execute()
nLeft = 0
for i = 1 to 30
        nLeft += StzFileExists(cDir + "/lesson" + i + ".txt")
next
? nLeft
#--> 30
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files, the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Trente ! Le modèle d'IA a demandé, a été refusé, a appelé `Execute()` malgré tout, et rien n'a changé sur le disque.

Le site montre cela avec un cours plus grand, dans sa démonstration d'éducation du 30 septembre 2026, que je n'ai pas exécutée pour ce livre, et il imprime un compte de six cent dix. Ce chiffre est celui du site, cité avec sa source et sa date. J'imprime trente, parce que trente est ce que ce bloc exécute.

Une personne peut discuter avec un plan. Le plan est un médium. Un relecteur rejette une étape, avec une raison. Que font les vingt-neuf autres étapes, et que reste-t-il sur le disque ? Le bloc imprime les comptes, non le nom du fichier :

HARO

```
oRev = oBench.GenerateUpdatePlan()
oRev.RejectOperation(2, "reviewer: keep lesson 2")
oRev.SetExecutor(PIActor("oncall"))
oScopeC = new stzCommitScope()
oScopeC.AllowUnder(cDir)
oRev.SetScope(oScopeC)
aRes = oRev.Execute()
? aRes[1][2]
#--> 29
? aRes[2][2]
#--> 1
```

```
nLeft = 0
for i = 1 to 30
        nLeft += StzFileExists(cDir + "/lesson" + i + ".txt")
next
? nLeft
#--> 1
StzDirDeleteAll(cDir)
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Un acteur nommé sous une portée valide les vingt-neuf autres (29) ; une est sautée (1) ; et un fichier reste sur le disque (1), celui de l'étape que le relecteur a protégée.

Une règle, deux couches, les mêmes mots. Le refus que le chapitre 17 a montré au graphe, et que le fichier a montré plus haut, est aussi appliqué dans la boucle de ticks propre au moteur, à l'enregistrement. Il l'est dans la phrase de la règle du graphe, no-llm-effectful. Un modèle d'IA qui a la capacité de produire des effets est refusé. Un modèle qui ne l'a pas s'enregistre comme tout autre emplacement, un emplacement étant la place que prend un agent dans la boucle :

HARO

```
StzResetEngineAgentLoop()
nL = stzenginezutercreate("summarizer")
? stzengineagentloopregister(nL, 1, 1, 0, 1, 0, 10, "summarises tickets")
#--> -6
? stzengineagentlooplastrefusal()
#--> AGENTLOOP-R6: agent 'summarizer' (slot 0) breaks the rule no-llm-effectful -- llm actor
holds 'effectful' -- an LLM proposes, only a pi-gate commits. Registration refuses it. Drop the
effectful capability, or register it as a pi-gated composition rather than an llm actor.
nL2 = stzenginezutercreate("proposer")
? stzengineagentloopregister(nL2, 1, 1, 0, 0, 0, 10, "proposes summaries")
#--> 0
StzResetEngineAgentLoop()
```

✓ exécuté : a affiché ce que ce bloc promet

Le moteur a refusé le résumeur, dans les mots de la règle du graphe ! Les arguments après le premier, qui est l'agent lui-même, sont, dans l'ordre où la garde les donne : le genre (1 est un modèle d'IA), la classe de réversibilité, le déclencheur et la

capacité. Ce qui reste est la priorité, l'intervalle en millisecondes et la phrase de couverture. Ce sont les noms de bas niveau du moteur, et je les montre pour que vous voyiez le refus venir du moteur, dans les mêmes mots.

Un agent de l'effectif propre du domaine de projets, répété. Jusqu'au 22 août 2026, un agent de l'effectif propre du domaine de projets, celui qui fait tourner le registre, déplaçait directement un vrai fichier de journal, sur un tick. Il n'avait ni plan, ni portée, ni acteur validant.

Un registre, ici, est un relevé de ce qui s'est passé, tenu dans l'ordre où cela s'est passé. Cet agent fait tourner le registre propre du domaine de projets. Le chapitre 21 a le registre de sécurité, que le tribunal écrit et jamais la chose qu'il juge, une chaîne de hachage où une ligne modifiée est nommée. Le chapitre 23 a le registre de traversées de la constellation, avec les durées. Le mot a un autre sens ailleurs : l'une des cinq formes que peut prendre une réponse, ou le registre des innovations de l'annexe A.

La doctrine avait été écrite avant d'être vraie, et le dossier l'a attrapé le jour même. Depuis, la fonction qui écrit pour cet agent a un chemin de répétition. Dans le tick d'un agent qui tient un atelier, elle écrit dans le jumeau. Hors d'un tel tick, dans l'état de migration que le domaine de projets vivant fait encore tourner, c'est toujours l'écriture directe. La déclaration ci-dessous est la mienne, écrite pour ce livre ; ce n'est pas le fichier d'effectif vivant, que je n'ai pas exécuté. Avec un atelier, l'agent déplace le fichier dans le jumeau. Deux compétences se déclenchent et vérifient ; le vrai fichier de journal est là où il était ; aucune archive n'existe ; le tick a exporté un plan. Et seul l'acteur validant, sous une portée, déplace le fichier :

HARO

```
cEstate = StzReplace(WorkingDirectory(), "\", "/") + "/_estate"
StzDirCreatePath(cEstate + "/dashboard")
StzDirCreatePath(cEstate + "/journal")
SetEnvVar("STZ_SOFTANZA_ROOT", cEstate)
StzFileWrite(cEstate + "/journal/2000-01-scar.md", "an old journal entry" + char(10))
StzFileWrite(cEstate + "/dashboard/SESSION-LOG.md", "# SESSION-LOG" + char(10) + char(10) +
"2000-01-15 09:00 | someone | long ago" + char(10))
StzAllowAgentFunction("RollRelocateMonths", "trusted")
StzAllowAgentFunction("RollWritePointer", "trusted")
cRoll = "pia: 2
name: ledger-roll
```

```
kind: pi
coverage: moves journal months before the current one into archive and leaves a pointer
reversibility: reversible
schedule:
  event: roll
governance:
  authority: delegated
  risks:
    relocate-evidence: 2
  grants:
    - relocate-evidence
skills:
  - name: relocate
    when: always
    does: ring:RollRelocateMonths
    verify: fact archive holds month
    effect: relocate-evidence
    posture: trusted
  - name: pointer
    when: fact archive holds month
    does: ring:RollWritePointer
    verify: fact ledger holds pointer
    effect: relocate-evidence
    posture: trusted
"
oRoll = StzAgentDeclarationQ(cRoll).ToAgent()
oRoll.GiveWorkbench()
? oRoll.Cycle()
#--> 2
? StzFileExists(cEstate + "/journal/2000-01-scar.md")
#--> 1
? StzFileExists(cEstate + "/journal/archive/2000-01-scar.md")
#--> 0
oRP = oRoll.GenerateUpdatePlan()
? oRP.NumberOfOperations()
#--> 7
? len(oRP.Risks()) >= 1
#--> 1
oRP.SetExecutor(PIActor("roll-committer"))
oScopeR = new stzCommitScope()
oScopeR.AllowUnder(cEstate)
oRP.SetScope(oScopeR)
oRP.Execute()
? StzFileExists(cEstate + "/journal/2000-01-scar.md")
#--> 0
? StzFileExists(cEstate + "/journal/archive/2000-01-scar.md")
#--> 1
StzDirDeleteAll(cEstate)
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le compte de sept opérations est ce que cette exécution mesure. La garde de la bibliothèque ne promet qu'au moins cinq.

Les blocs ont montré ce que fait le monde ; la section 4 dit ce que j'ai refusé de construire à la place.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé les agents sûrs. La raison de la doctrine est que la sûreté du terrain essaie de rendre digne de confiance un acteur fondamentalement imprévisible. Je tiens cette raison pour une vue, non pour un résultat. L'image de la doctrine est que demander si l'agent d'IA pourrait s'échapper, c'est comme demander si un moteur d'échecs pourrait déplacer une pièce sur un échiquier physique. Elle dit que la surface d'outils de l'agent ne contient aucune opération qui touche le disque, une base de données ou le réseau. C'est l'affirmation de la doctrine, et la section 6 en donne les limites.

J'ai refusé un contrôle de permission comme frontière. Ce n'est pas un contrôle que l'agent ne pourrait pas contourner. Les pouvoirs sont séparés : la créativité sans l'autorité, l'autorité sans la créativité.

J'ai refusé un second atelier pour les machines. Le bac à sable est le même atelier que le programmeur humain utilise déjà. Le cadre a été conçu pour des programmeurs humains, et tout ce qui est conçu pour en libérer un est exactement ce qu'il faut pour en contenir un artificiel.

J'ai refusé de laisser la barrière facultative. La décision du 22 août 2026 dit que la barrière d'enregistrement est la doctrine, et que l'adhésion volontaire est un état de migration, non le dessin. Le dossier est honnête sur la découverte derrière la décision : là où la doctrine des agents qui ne peuvent pas vous nuire s'imposait ce jour-là, il n'y avait rien. Le monde sûr existait, et le répertoire agentique n'en mentionnait rien. La décision l'a refermé le jour même.

J'ai refusé un bouchon. Demander un agent d'IA réservé lève une erreur. L'architecture dit que c'est la loi qui travaille, et non un défaut.

J'ai refusé d'abandonner en silence. Dans le moteur, un envoi vers une boîte de réception pleine est refusé avec un code et compté. C'est testé dans le code Zig du moteur, par un test que je n'ai pas exécuté, et aucun bloc ici ni aucune garde narrée

ne le montre. Quand la file d'exécution est pleine, les visites suivantes sont abandonnées et comptées. La boucle dit aussi ce qu'elle ne peut pas voir : l'intérieur d'un rappel, l'horloge, et si une déclaration de couverture est vraie.

J'ai refusé de laisser une machine être le dernier recours. La constitution de l'interface du domaine de projets a un droit, le sixième sur six : le Droit à un humain. Partout où une machine sert, une personne est joignable, clairement, et sans pénalité. Il est appliqué par un jugement, celui d'une personne, et non par une garde qui s'exécute ; la bibliothèque ne le mentionne pas. La constitution le tient pour le bout humain de la boucle : c'est la règle 102 de la constitution (annexe C).

La section 5 nomme les gardes qui prouvent ce que les blocs ont montré.

5. LE TRIBUNAL

`safeworld_narrated` a sept scènes : la réversibilité comme sixième contrat ; la barrière d'enregistrement dans les phrases du moteur ; la table de la posture face à la réversibilité ; le tribunal du fichier au chargement. Trois autres suivent : ce que le fichier déclare, qui atterrit dans la gouvernance de l'agent ; l'atelier, du disque qui n'a pas bougé jusqu'à la piste d'audit ; et l'agent de l'effectif répété puis validé. `agentfile_narrated` prouve qu'un agent est un fichier jugé au chargement, avec un refus par barrière, et qu'un répertoire est un déploiement. `agentloop_engine_narrated` prouve que la boucle du moteur énonce sa couverture à l'exécution, que l'enregistrement est une barrière, qu'une règle donne les mêmes mots dans deux couches, et qu'une file trop pleine abandonne des visites et les compte. `containment_drill_narrated` prouve un exercice contre une vraie cible en boucle locale : un enquêteur modèle propose le confinement et ne valide rien ; un humain valide, et le confinement tient quand on le vérifie de l'extérieur. Dans mon exécution du 8 octobre 2026, l'exercice a lu 443 ms pour détecter et 61 ms pour contenir, sur cette machine. Les données du livre ne contiennent pas cette exécution. Le site imprime d'autres nombres tirés de sa propre exécution, et le modèle de menace d'autres encore tirés de la sienne. Les trois diffèrent parce que l'exercice mesure une horloge. La limite de l'exercice est une seule forme d'attaque,

en boucle locale, sur une seule machine, lancée à la demande. `virtual_system_twin_narrated` prouve le jumeau : une répétition ne touche rien de réel ; le plan se narre et signale ses risques ; une portée refuse ; un relecteur rejette une étape ; les instantanés reviennent en arrière.

La garde `roster_narrated` n'est pas dans l'en-tête, parce qu'elle ne peut pas s'exécuter depuis un instantané de la bibliothèque : elle charge les fichiers d'effectif du domaine de projets depuis un dépôt voisin. La répétition de l'agent de l'effectif est prouvée par `safeworld_narrated`, dont la septième scène est le dernier bloc ci-dessus.

La section 6 se tourne vers le terrain, et vers où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`safeworld_narrated` 76 assertions
base/test/agentic/safeworld_narrated.ring

`agentfile_narrated` 88 assertions
base/test/agentic/agentfile_narrated.ring

`agentloop_engine_narrated` 54 assertions
base/test/agentic/agentloop_engine_narrated.ring

`containment_drill_narrated` 27 assertions
base/test/security/containment_drill_narrated.ring

`virtual_system_twin_narrated` 48 assertions
base/test/system/virtual_system_twin_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

La table propre du dossier est honnête, et j'en porte les lignes en prose.

Le terrain mène en connecteurs : messagerie, disques et gestionnaires de tickets, magasins de vecteurs hébergés, interfaces en flux. Ils sont atteignables depuis Softanza et non pavés.

Le terrain mène en recherche sur des corpus ouverts. Softanza gagne sur un domaine sélectionné et perd sur le web ouvert.

Le terrain mène en raisonnement de frontière. Cette pile orchestre les plus grands modèles à chaque appel.

Le terrain mène en écosystème et en recrutement. Il y a un marché du travail pour cette pile, et aucun pour celle-ci.

Quand choisir laquelle, d'après la narration : la pile courante, quand la tâche est un travail de langage ouvert sur une matière que vous ne pouvez pas sélectionner. Ses exemples sont résumer des documents arbitraires, des produits conversationnels, de la recherche sur le web ouvert. La couche Softanza, quand les actions ont des conséquences et que quelqu'un finira par demander pourquoi elle a fait cela. Parmi ces cas, dans les mots de la narration : le travail réglementé ou audité, tout ce qui tourne sans surveillance, et tout ce qui doit continuer à fonctionner hors ligne à coût marginal nul. Les deux ne s'excluent pas, et un modèle d'IA peut siéger dans un système Softanza comme proposeur.

Le jumeau est un modèle de la réalité, non la réalité. Le cadre demande de la fidélité et de l'honnêteté, et le plan se revalide contre la réalité avant d'être validé. Une fois le vrai fichier supprimé dans le dos du jumeau, la vérification dit que la cible est absente dans la réalité. Je l'ai vue avertir, et je n'ai pas testé si elle bloque la validation.

Et la plus grande perte honnête. Le modèle de menace met *le code déjà en cours d'exécution dans le processus* hors de portée : il peut faire tout ce que le processus peut faire, effacer le registre ou lire un secret. L'atelier contient ce qui est écrit à travers l'atelier. Une fonction qui ne demande pas l'atelier n'est pas confinée. Les lectures sont libres : l'en-tête de l'atelier lui-même dit que les lectures sont de la

perception et restent libres, et que seules les écritures sont confinées au jumeau. L'architecture dit que l'évasion n'est pas un défaut à fermer en le supprimant. Elle la réduit par une posture déclarée, et par du code, non le fichier, qui décide quelles fonctions un agent peut appeler. Je n'ai pas exécuté de fonction de confiance qui contourne l'atelier : j'ai lu cette limite dans l'architecture, dans le modèle de menace et dans l'en-tête de l'atelier.

7. LA LIMITE

CONSTRUIT	la réversibilité comme contrat déclaré et comme barrière d'enregistrement ; la posture confrontée à la réversibilité, à la porte et au chargement ; l'atelier, le plan, le rejet du relecteur, l'acteur validant sous une portée ; l'exercice de confinement ; les emplacements bornés et la file du moteur (la boîte de réception bornée n'est testée qu'en Zig, non exécutée pour le livre)
CONÇU	le côté de l'agent dans la négociation autour d'un plan ; la salle d'examen ; les branches comme esprits parallèles ; les jumeaux pour les bases de données, les interfaces et les réseaux
ABSENT	un plan sans exécutant qui soit refusé ; un classement des risques d'un plan ; le code déjà en cours d'exécution dans le processus, qui est hors de portée ; le Droit à un humain comme contrôle machine

Ce qui est construit : la réversibilité comme contrat déclaré, et la barrière d'enregistrement, dans la porte de l'ancien langage hôte et dans le moteur ; la posture confrontée à la réversibilité à la porte et au chargement. L'atelier est construit, avec un plan qui se narre, signale, valide, accepte un rejet et tient une piste d'audit ; l'agent de l'effectif répété puis validé l'est aussi, et l'exercice aussi. Les structures bornées du moteur sont construites : 64 emplacements d'agent, une boîte de réception de 8 messages (testée en Zig seulement, non exécutée pour ce livre) et une file d'exécution de 256.

Ce qui est conçu : le côté de l'agent dans la négociation, où le plan parvient à l'agent mais où la conversation sur le plan ne le fait pas encore. L'architecture le dit, et le modèle de menace le répète. J'ai cherché dans le code de la bibliothèque et n'ai trouvé aucune révision d'un plan. La salle d'examen, qui certifierait un agent pour

une portée de validation comme on certifie un conducteur pour une catégorie de permis : elle apparaît dans des documents et dans aucun code. Les branches comme esprits parallèles : les instantanés et le retour en arrière sont construits, et je n'ai trouvé aucun appel de branche. Les jumeaux pour les bases de données, les interfaces et les réseaux : le cadre les nomme. Le jumeau des fichiers est construit, et le jumeau de l'environnement du processus aussi, avec ses variables, son répertoire de travail et son lancement mis en file (chapitre 24). Les trois autres ne le sont pas.

Chaque point ci-dessous est un écart, trouvé en exécutant et en lisant.

- **Un exécutant non défini et une portée non définie ne sont pas gardés.** Un plan sans exécutant répond qu'aucun exécutant n'est défini, qu'un humain au clavier n'est pas gardé, et `Execute()` est permis. Le refus du bloc ci-dessus s'applique à un modèle d'IA désigné comme exécutant. De même, le contrôle de portée ne s'exécute que lorsqu'une portée est définie : un plan avec un exécutant et sans portée est validé sans aucun contrôle des chemins qu'il touche. Donc « seul un acteur gouverné valide » est exact quand un exécutant et une portée sont définis, et une personne au clavier peut valider un plan qui n'en nomme aucun.
- **Un plan signale ; il ne classe pas.** La narration dit qu'un plan classe ses propres risques. Le code renvoie un signal par étape destructrice, dans l'ordre des opérations, avec une étiquette comme DESTRUCTIVE. Il n'y a pas de score ni d'ordre de gravité. Le chapitre dit « signale ».
- **La barrière est facultative chez l'hôte.** Un programme qui héberge la bibliothèque doit déclarer qu'il emploie la boucle du moteur ; tant que le dernier de ces programmes dans l'arbre propre de la bibliothèque n'a pas déclaré, le défaut reste inactif, et un programme neuf fait tourner l'ancienne boucle de tick de l'ancien langage hôte.
- **La boucle vérifie qu'une déclaration de couverture existe, non qu'elle est vraie.**
- **Les limites de l'exercice :** une forme d'attaque, en boucle locale, sur une seule machine.

Six portes suivent, le même chapitre vu de six endroits.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Votre agent est un fichier et quelques fonctions. Il s'exécute et croit avoir agi, et le disque ne bouge pas. Vous lisez le plan, vous choisissez l'acteur validant et la portée, et le tribunal vous dit en une phrase quelle règle vous a refusé.
L'analyste	Chaque proposition est un plan narré d'opérations avec les étapes destructrices signalées, et une ligne d'audit pour chaque validation. Vous pouvez lire ce que l'agent voulait faire, étape par étape, sans l'exécuter. Les signaux sont par genre, non notés.
Le concepteur	Le plan est l'unique artefact qui passe de l'agent à la personne, et une personne peut rejeter une étape. Le Droit à un humain dit qu'une personne est joignable partout où une machine sert, et le droit est tenu par un jugement.
Le décideur	Vous déclarez ce que chaque agent couvre et à quel point il est réversible avant qu'il puisse exister, et vous décidez la portée et l'acteur qui valide. Un modèle d'IA est structurellement incapable de valider, même trompé ; le modèle de menace suppose qu'il le sera. La partie la plus difficile est celle que le dessin vous laisse : quelles fonctions le code permet à un agent d'appeler.
Le formateur	L'agent d'un apprenant qui range un répertoire de cours en supprimant tout est une leçon sur ce qu'est un plan. Elle voit ce qu'il aurait fait, et les fichiers sont toujours là.
L'agent	Vous pouvez répéter sans limite, à l'intérieur du jumeau. Votre seul export est un plan, et on peut discuter avec vous à son sujet. Vous ne pouvez pas valider. Aujourd'hui la discussion est à sens unique : vous ne recevez pas encore l'objection du relecteur pour réviser.

9. EXERCICE

Choisissez un répertoire de dix fichiers que vous pouvez perdre. Écrivez un fichier d'agent pour un travail de rangement sur lui : ce qu'il couvre, à quel point il est réversible, une compétence. Avant d'exécuter quoi que ce soit, écrivez sous chaque ligne la promesse que vous attendez : combien de fichiers existent sur le disque après un cycle, combien d'opérations le plan contient, et combien d'entre elles le plan signale comme des risques. Puis écrivez ce qui arrive quand vous désignez un modèle de langage comme exécutant.

HARO

```
cDirMine = StzReplace(WorkingDirectory(), "\", "/") + "/_mine"
StzDirCreatePath(cDirMine)
for i = 1 to 10
        StzFileWrite(cDirMine + "/file" + i + ".txt", "file " + i)
next
oMine = new stzVirtualFileSystem()
oMine.SetActor("my-tidy-bot")
# your tidy-up proposal here, with the promise you expect under each line
StzDirDeleteAll(cDirMine)
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Puis donnez le plan à un acteur validant sous une portée qui ne couvre pas le répertoire, et écrivez la promesse de nouveau. Si une promesse se rompt, ne changez pas la promesse : expliquez d'abord la sortie. Un bon apprenant est surpris une fois. Le plan signale ses étapes destructrices et ne les classe pas, et un plan sans aucun exécutant est permis.

Je vous laisse celui-ci, exprès. Prenez le bloc qui a validé la liste de prix. Les six lignes qui ont construit `oScope3` et validé sous elle deviennent ces trois-ci, et les lignes qui les suivent restent :

HARO

```
oPlan3 = oAg.GenerateUpdatePlan()
oPlan3.SetExecutor(PIActor("shop-committer"))
oPlan3.Execute()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le, et demandez au disque ce que le plan a fait, puis demandez-le à la piste d'audit. Je ne vous dis pas ce que vous trouverez. La page dit pourquoi à la section 7, dans le premier point de sa liste d'écarts ; lisez-le après avoir regardé, non avant. Quand vous saurez dire pourquoi « seul un acteur gouverné valide » tient pour ce plan ou ne tient pas, vous saurez de quoi cette phrase dépend.

Le chapitre 21 porte sur le tribunal lui-même, sous quatre de ses formes : promesses, perception, sécurité et performance. Son principe est qu'une affirmation ne vaut que ce que vaut le tribunal qui peut la refuser, et que là où aucun tribunal ne voit, le

dossier le dit.

OÙ SONT LES PREUVES

NARRATION	stz-agents-that-cannot-hurt-you-narration.md
NARRATION	stz-service-virtualization-code-first-subscribe-later.md
SUR LE SITE	agentic
SUR LE SITE	security
DÉCISIONS DE L'ÉTUDE	12.18, 12.19, 12.22, 12.23, 12.29

PARTIE IV · CHAPITRE 21

Le tribunal : promesses, perception, sécurité et performance

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 12:59 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 7 bloc(s) sur 7 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je reprends le tribunal du chapitre 5, qui juge quiconque a écrit le code. J'en montre quatre, chacun par quelque chose qu'il refuse : les promesses, la perception, la sécurité et la performance. Je fais parcourir au voyageur un fichier de trois promesses, et je demande au dossier combien des huit traditions musicales une personne a entendues. Je modifie un registre scellé et je lis quelle entrée la chaîne nomme, et je laisse un modèle d'IA proposer un plan de performance qu'il ne peut pas valider. Je dis ce que j'ai refusé, sept choses en tout, dont le mot « narration » pour ce qui ne peut pas mentir, une seconde bibliothèque d'assertions, un sixième état pour un arrêt, une garantie sans garde. Le principe tient les quatre tribunaux ensemble : une affirmation ne vaut que ce que vaut le tribunal qui peut la refuser, et là où aucun tribunal ne voit, le dossier le dit.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Quatre pratiques du terrain sont quatre tribunaux, et on les voit rarement comme un seul.

La première est l'exemple qui s'exécute. Le doctest de Python exécute la session interactive trouvée dans une docstring et compare le texte imprimé. Rust exécute les exemples de sa documentation, et Go a des exemples testables avec une sortie

attendue dans un commentaire. L'exemple est la spécification. Sa faiblesse est que la comparaison est du texte contre du texte, et que c'est le harnais qui décide ce qu'est un échec.

La deuxième est le test d'instantané. Une capture d'écran ou un arbre rendu est stocké et comparé à l'exécution suivante. Il prouve que l'image d'aujourd'hui égale celle d'hier.

La troisième est l'audit contre une liste. Les listes du terrain sont celles de l'OWASP, du NIST et des cadres de conformité. Pour la détection et la preuve, la chaîne d'outils est l'agrégateur de journaux, le format de règles, l'orchestration des réponses.

La quatrième est l'instrument. Micrometer, OpenTelemetry, Prometheus et les profileurs de chaque runtime mesurent un programme de l'extérieur. Les quatre sont de la connaissance générale, et non une étude du dossier, sauf là où je nomme le fichier.

2. LE PREMIER PRINCIPE

Une affirmation ne vaut que ce que vaut le tribunal qui peut la refuser. Et là où aucun tribunal ne voit, le dossier le dit.

Le dossier la tient en deux phrases qui se rejoignent. Le modèle de menace : chaque garantie nomme la garde qui la prouve, et une garantie sans garde est listée comme un risque ouvert. Et la loi de la perception : une comparaison prouve l'identité et jamais la justesse, et si une personne est le dernier consommateur, une personne est la dernière barrière. Sous les deux se trouve la doctrine que la bibliothèque répète dans chaque volet : l'expression est libre, et l'admission est gouvernée.

3. LA FORME SOFTANZA

Le chapitre 5 a parlé du tribunal qui juge quiconque a écrit le code. Ce chapitre a quatre tribunaux, un par genre d'affirmation. Je montre chacun par quelque chose qu'il refuse.

LE TRIBUNAL DES PROMESSES

Une promesse, dans ce livre, est la ligne `#-->` sous une expression, et c'est une affirmation sur un arrêt du programme. L'outil qui la lit, appelé le voyageur (il appartient au volet nommé Testoor), dit qu'une promesse est tenue, divergente, non atteinte ou non jugée. Une promesse qui se dispute avec elle-même, de la prose après une valeur, n'est jamais accusée. Que dit le voyageur d'une promesse vraie, d'une promesse fausse et d'une promesse qui se dispute avec elle-même ? Exécutez-le :

HARO

```
write("tour.haro", '? 1 + 1' + nl + '#--> 2' + nl +
        '? 1 + 1' + nl + '#--> 3' + nl +
        '? "C"  #--> "C" but should be "sm_AS"' + nl)
oTour = TravellerQ().Silent().Walk("tour.haro")
aClaims = oTour.Records()[1][:run][:stops][1][:claims]
? aClaims[1][:state]
#--> kept
? aClaims[2][:state]
#--> diverged
? aClaims[3][:state]
#--> unjudged
? oTour.ExitCode()
#--> 1
remove("tour.haro")
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Il dit `kept`, `diverged` et `unjudged`, et le code de sortie est `1`. Ce code est celui du lanceur, et non celui du langage.

Une promesse peut être que la ligne lève une erreur. Cela se lit comme une levée d'erreur promise. Quand elle survient, comment la tournée finit-elle ? Exécutez-le :

HARO

```
write("raise.haro", '? "before"  #--> before' + nl +
        '? Boom()  #--> ERROR: boom happened' + nl +
        'func Boom()' + nl + char(9) + 'raise("boom happened")' + nl)
oRaise = TravellerQ().Silent().Walk("raise.haro")
aRun = oRaise.Records()[1][:run]
? aRun[:stops][1][:claims][2][:state]
#--> kept
```

```
? aRun[:end]
#--> finished
? oRaise.ExitCode()
#--> 0
remove("raise.haro")
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

La promesse est `kept`, la fin est `finished` et le code de sortie est `0` ! Le processus du langage s'est terminé sur une erreur, et la tournée a fini comme elle avait dit qu'elle le ferait.

Chaque règle de ce juge a été apprise en regardant le harnais accuser la bibliothèque à tort. L'en-tête de la garde le dit. Une promesse courte `6` s'est un jour cachée à l'intérieur d'un `[ 6, 28 ]` plus loin ; c'est pourquoi il existe une règle qu'un mensonge court ne peut pas se cacher dans une liste plus loin, à côté de la règle sur les graphies qu'une valeur vraie peut prendre. Le juge propre à ce livre est une copie du juge du site et non du voyageur. Il a moins de règles que le voyageur, et celle qu'il garde est celle sur laquelle ce livre repose : la construction échoue quand un bloc ne tient pas sa promesse.

LE TRIBUNAL DE LA PERCEPTION

La phrase du dossier est qu'une comparaison prouve l'identité, et qu'elle ne prouve jamais la justesse. La bibliothèque tient le cas. Un tracé en barres dessinait des barres verticales pour des données horizontales pendant que tous ses soixante-treize exemples continuaient de passer. C'est pourquoi les tracés comparent désormais l'image exactement. Cela rend l'identité stricte ; cela n'en fait pas la justesse. La justesse est une personne.

Donc un monde déclaré porte son état de perception. Huit traditions musicales sont déclarées comme données dans la bibliothèque. Combien d'entre elles une personne a-t-elle entendues ? Demandez au dossier :

HARO

```
nUnheard = 0
for cName in StzSoundUniverses()
```

```
        if StzSoundUniverseQ(cName).Listener() = "UNPERCEIVED"
                nUnheard++
        ok
next
? len(StzSoundUniverses())
#--> 8
? nUnheard
#--> 7
? left(StzSoundUniverseQ("tunisian").Listener(), 5)
#--> HEARD
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond 8, 7 et HEARD : huit déclarées, et sept d'entre elles non perçues ! Une seule est entendue, et le dossier de celle-ci porte un nom, une date et un verdict. Le compteur de ce bloc ne fait que compter les non entendues ; aucun compteur ne décide du verdict. L'unique tradition qui a été entendue l'a été par moi le 26 septembre 2026, à travers cinq exemples originaux, et mon verdict a été « far from being qualified ». Laquelle des parties échoue, l'accord, le phrasé, le rythme ou le timbre, je ne l'ai pas encore dit. Ces mots sont les miens, et ils sont l'unique perception enregistrée qui soit un refus.

La loi est celle du domaine de projets, c'est-à-dire de l'ensemble de mes projets, adoptée le 25 septembre 2026 sur mon instruction. Elle exige que le dossier nomme celui qui perçoit et cite le verdict. Sa troisième exigence est celle que j'emporte avec moi : non perçu est un état, pas un échec. Elle refuse aussi deux choses. Elle ne dit pas qu'un humain doit garder chaque commit, parce qu'elle lie à la granularité d'une phase. Et elle ne dit pas qu'elle se généralise à partir d'un volet sans preuve supplémentaire, et elle a été retenue un mois pour ce motif.

La galerie d'images est l'endroit où elle n'est pas satisfaite. La bibliothèque tient quatre-vingt-quinze jugements d'images et de sections de texte, dans trois fichiers d'index, commités le 5 octobre 2026. Chacun est d'un modèle d'IA, lisant l'image ou lisant le texte que le programme a imprimé. Aucun ne porte de nom humain. Le premier index le dit en une phrase : aucun humain n'a regardé ces images. Le registre des constats se termine par une ligne que je répète : un verdict humain sur un échantillon est encore dû. Le dossier ne tient aucun verdict humain sur une image de la galerie. Elle reste non perçue.

LE TRIBUNAL DE LA SÉCURITÉ

Le volet de sécurité part d'une affirmation. Le modèle de menace suppose qu'un modèle de langage détourné passera, et le rend inoffensif. Le modèle d'IA ne détient jamais la capacité, et donc un modèle d'IA trompé ne peut toujours pas agir. Le document n'essaie pas de détecter chaque invite malveillante.

La preuve de ce qui s'est passé est une chaîne de hachage. Modifiez n'importe quelle ligne après la première, et la chaîne dit quelle entrée s'est rompue. Si j'en modifie une, quelle entrée nomme-t-elle, et une mauvaise clé déclenche-t-elle la même accusation ? Exécutez-le :

HARO

```
oLed = StzSecurityLedger(256)
oLed {
        Record(StzSecurityRefusal("auth.login.failed", HumanActor("admin"), "user:admin", "wrong
credentials"))
        Record(StzSecurityRefusal("secret.reveal.refused", LLMActor("advisor"), "secret:payments-
key", "actor is not effectful"))
        SealTo("evidence.txt", "the-evidence-key")
}
? StzVerifySealedLedger("evidence.txt", "the-evidence-key")[:why]
#--> chain intact over 2 entries
? StzVerifySealedLedger("evidence.txt", "another-key")[:why]
#--> the chain is intact but the SEAL does not match this key
write("evidence.txt", StzReplace(read("evidence.txt"), "actor is not effectful", "routine
maintenance"))
? StzVerifySealedLedger("evidence.txt", "the-evidence-key")[:why]
#--> the chain breaks at entry 2 -- that record (or one before it) was edited
remove("evidence.txt")
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le vérificateur répond `chain intact over 2 entries`, puis `the SEAL does not match this key`, puis `the chain breaks at entry 2`. La chaîne a nommé la rupture ! Une mauvaise clé et une ligne modifiée sont deux accusations différentes. Remarquez quelle entrée j'ai choisi de modifier : la deuxième. Une modification de la première entrée n'est pas détectée. Le vérificateur contrôle le texte de chaque entrée contre l'empreinte de l'entrée qui la précède, et la première n'en a aucune avant elle.

Cela a caché ma première tentative. La narration énonce la limite dans ses propres mots : ce n'est pas une protection contre un attaquant qui exécute déjà du code dans le processus.

Un secret est la même loi appliquée à une clé. Son seul visage public est son descripteur. Une main qui peut valider peut le lire. Que répond le tribunal quand un modèle de langage demande ? Exécutez-le :

HARO

```
StzOpenSecurityLedger(64)
oKey = new stzSecret("deploy-key")
oKey {
        FromLiteral("ssh-ed25519-AAAA")
        ? Descriptor()
        #--> <secret 'deploy-key' (secret) from literal>
        ? Reveal(HumanActor("ops"))
        #--> ssh-ed25519-AAAA
}
try
        oKey.Reveal(LLMActor("planner"))
catch
        ? cCatchError
done
#--> Refused: actor 'planner' (posture sandboxed) may not reveal secret 'deploy-key'. Only an
effectful, non-sandboxed actor can read a secret.
? len(StzSecurityLedgerQ().Refusals())
#--> 1
StzCloseSecurityLedger()
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `Refused: actor 'planner' (posture sandboxed) may not reveal secret 'deploy-key'`. Le tribunal a refusé le modèle d'IA, et le registre compte `1` refus, alors qu'aucune ligne du bloc n'y enregistre rien ! C'est la loi du chapitre 17, qu'un modèle ne détient pas la capacité de produire des effets, appliquée à une clé, avec le dossier en plus.

Qu'est-ce que la table de garanties lie à une garde ? La méta-garde du modèle de menace vérifie que le document existe avec ses sections, et que chaque ligne de garantie contient une preuve. Une preuve est le chemin d'une garde, une exécution de fuzzing, un contrôle de nomenclature logicielle ou un manuel d'exploitation. Elle

vérifie que chaque chemin de garde dans les lignes existe sur le disque, et qu'une fausse ligne plantée, avec une garantie sans preuve, est attrapée. Elle n'exécute pas les gardes, et elle ne vérifie pas qu'une garde affirme ce que dit sa ligne. Cela, c'est le travail de la barrière de sécurité. Donc « chacune liée à une garde » est, exactement, « chacune nomme au moins une preuve, et tout fichier de garde qu'elle nomme existe ».

À ce commit, la table tient 43 garanties. Le site dit 38, sur trois pages. Les deux sont vrais de leurs dates : le document avait 38 lignes le 30 septembre 2026, et les lignes de 39 à 43 ont été ajoutées le 5 octobre. Le nombre du site est au site de le mettre à jour.

LE TRIBUNAL DE LA PERFORMANCE

Un instrument doit un nombre. Le chronomètre honnête donne un nombre issu d'une horloge qui ne va que vers l'avant. Il exclut une pause, et il expédie deux spans d'une même trace dans un seul lot pour un collecteur d'observabilité. Si je le mets en pause 200 millisecondes, la pause compte-t-elle dans le temps écoulé ? Exécutez-le :

HARO

```
w = StzStopwatchXT("import-orders")
w {
        Lap("parsed")
        Pause()
        StzEngineTimeSleepMs(200)
        Resume()
        Stop()
        ? isnumber(ElapsedMs())
        #--> 1
        ? ElapsedMs() < 200
        #--> 1
        ? Record()[:running]
        #--> 0
}
w2 = StzStopwatchXT("notify-customer")
w2.JoinTrace(w.TraceParent())
w2.Stop()
? w2.TraceId() = w.TraceId()
#--> 1
oBatch = StzOtelBatch("shop")
oBatch.AddSpan(w)
oBatch.AddSpan(w2)
```

```
? Q(oBatch.ToJson()).NumberOfOccurrence(w.TraceId())
#--> 2
```

✓ exécuté : a affiché ce que ce bloc promet

La pause n'a pas compté : dans la scène du chronomètre, `ElapsedMs() < 200` a imprimé `1`, et `Record()[:running]` a imprimé `0`, parce que le chronomètre s'était arrêté. Le second chronomètre a rejoint la trace du premier : `w2.TraceId() = w.TraceId()` a imprimé `1`, et le JSON du lot contient l'identifiant de trace 2 fois, pour deux spans.

Prudence sur ce que cela ne prétend pas. La narration dit que le chronomètre rejoint reçoit un nouveau span. J'ai vérifié : il reçoit la même trace, comme le montre le `1` ci-dessus, et le même identifiant de span, que ce bloc n'imprime pas. La phrase tient pour l'écho du serveur, que je n'ai pas exécuté, et non pour le chronomètre. C'est une demande à la bibliothèque.

L'instrument n'est pas la fin du tribunal. Le jugement et l'action vivent à l'intérieur du programme. Un plan de performance est proposé par une machine, et seul un acteur effectif le valide. Que se passe-t-il quand un modèle d'IA demande à le valider, puis quand une personne le fait ? Exécutez-le :

HARO

```
oPlan = StzPerfPlan("relieve-queue")
oFleet = new FakeFleet
oPlan {
        Propose(:ScaleUp, "nlp", "queue.depth: measured 900 (must be <= 100)")
        ? MayCommit(LLMActor("advisor"))
        #--> 0
        ? ExecuteOn(oFleet, LLMActor("advisor"))
        #--> 0
        ? oFleet.CallCount()
        #--> 0
        ? ExecuteOn(oFleet, HumanActor("mansour"))
        #--> 1
        ? oFleet.CallCount()
        #--> 1
        Show()
        #--> Perf plan relieve-queue -- 1 proposed action(s).
        #      1. scaleup 'nlp' -- queue.depth: measured 900 (must be <= 100)
        #    Audit (2):
        #      #1 REFUSED scaleup 'nlp' by advisor -- actor is not effectful -- expression is
```

```
free; admission is governed
        #       #1 COMMITTED scaleup 'nlp' by mansour -- queue.depth: measured 900 (must be <=
100)
}

class FakeFleet
        aFleetCalls = []
        def ScaleUp(pcTag)
                aFleetCalls + [ "scaleup", pcTag ]
        def CallCount()
                return len(aFleetCalls)
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le parc, ici, est un double de répétition, tout objet qui répond aux mêmes verbes effectifs. La boucle est prouvée contre le double et non contre un vrai parc. Un modèle qui demande à valider est refusé avant toute tentative : `ExecuteOn` a imprimé `0` et `CallCount` a imprimé `0`, donc le double a compté zéro appel ! Pour la personne, `ExecuteOn` a imprimé `1` et `CallCount` a suivi avec `1`. La même action est jugée deux fois, et l'audit garde les deux jugements : `REFUSED scaleup 'nlp' by advisor -- actor is not effectful -- expression is free; admission is governed`, et `COMMITTED scaleup 'nlp' by mansour`.

Les quatre tribunaux ont parlé, et maintenant je dis ce que j'ai refusé.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé le mot pour la chose qu'il n'était pas. La structure de ce livre et le bilan de clôture de l'étude appellent les articles des « narrations qui ne peuvent pas mentir ». Ma propre décision du 7 octobre 2026 dit que « narration » nomme les articles sous le dossier de documentation de la bibliothèque, et non les suites exécutables. Le lanceur du site annote un article et ne décide plus s'il est montré. Le propre dossier du site du 8 octobre 2026, exécuté sur un autre commit que celui de ce livre, dit que sur 135 articles 55 se sont exécutés. Sur les 423 blocs de ceux-ci, 123 ont tenu leur promesse. Ce compte est celui du lanceur du site, sur des articles au commit du site, et non un compte des gardes de la bibliothèque. Donc les articles peuvent dériver. Ce qui ne peut pas mentir, ce sont les gardes exécutables, et la construction de ce livre, qui échoue quand un bloc ne tient pas sa promesse.

J'ai refusé une seconde bibliothèque d'assertions. Les décisions du lanceur de promesses refusent une seconde bibliothèque, une réécriture des fichiers de test, et un pourcentage de couverture avant qu'il y ait des routes. Elles refusent un double écrit pour la production d'une plateforme, et le mot « vérifié » sur une affirmation perceptive sans un nom et une phrase.

J'ai refusé de plier non jugé ou non perçu en réussite ou échec. Un arrêt a cinq états et pas de sixième : tenue, divergente, non atteinte, non jugée, non perçue. Un lanceur qui plie non jugé ou non perçu en réussite ou échec a tort. Je tiens cela pour une vue, non pour un résultat.

J'ai refusé d'analyser une preuve altérée. Un outil qui vérifie, avertit, puis analyse quand même apprend à ses utilisateurs à ignorer l'avertissement.

J'ai refusé de tout détecter. Les détections sont déclarées. La sentinelle ne parle que sur les transitions : silence, une alarme après cinq mauvais mots de passe, et rien de nouveau après un sixième, puisque l'histoire n'a pas redémarré. Elle n'écrit pas ses propres déclenchements dans le registre, parce que des détections sur des événements de détection invitent des boucles de rétroaction.

J'ai refusé de concurrencer l'écosystème du terrain. L'étude de performance du 29 juillet 2026 dit que le volet ne concurrence pas Grafana, Datadog ni l'écosystème d'OpenTelemetry, et ne devrait pas feindre de le faire. Il exporte.

J'ai refusé une garantie sans garde. La dernière règle du modèle de menace est qu'une garantie n'est ajoutée qu'avec sa garde.

Voyons maintenant les gardes qui prouvent ce qui s'est exécuté.

5. LE TRIBUNAL

`testoor_promises_narrated` prouve chaque règle de promesse plantée dans les deux sens. Un mensonge est divergent ; deux mensonges sont tous deux attrapés ; un mensonge court ne se cache pas dans une liste plus loin ; les graphies d'une valeur vraie sont toutes tenues. Une levée d'erreur attendue est honorée ; la prose est non jugée ; un programme qui ne compile pas n'est pas divergent ; une levée d'erreur en cours de route laisse les promesses suivantes non atteintes. `threat_model_narrated` prouve la méta-garde décrite plus haut, y compris son pendant négatif. `security_ledger_narrated`, `secretstore_narrated` et `secret_narrated` prouvent la chaîne et le sceau, le magasin et le secret. `security_sentinel_narrated` prouve la sentinelle déclenchée sur les fronts. `stopwatch_narrated` et `tracing_tail_narrated` prouvent l'instrument et le lot. `governed_loop_narrated` prouve le plan, le contrôle préalable, la validation et l'audit.

La construction compte 11 assertions dans `threat_model_narrated`, un compte statique de ses appels d'assertion. Les autres sont aussi comptées par la construction, et non exécutées ici. Le volet de performance a douze gardes. Je n'imprime pas leur somme, parce que la construction les compte.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`testoor_promises_narrated` 46 assertions
base/test/testoor/testoor_promises_narrated.ring

`threat_model_narrated` 11 assertions
base/test/security/threat_model_narrated.ring

`security_ledger_narrated` 35 assertions
base/test/system/security_ledger_narrated.ring

`secretstore_narrated` 26 assertions
base/test/system/secretstore_narrated.ring

`secret_narrated` 53 assertions base/test/system/secret_narrated.ring

`stopwatch_narrated` 35 assertions
base/test/perf/stopwatch_narrated.ring

`tracing_tail_narrated` 26 assertions
base/test/perf/tracing_tail_narrated.ring

`governed_loop_narrated` 31 assertions
base/test/perf/governed_loop_narrated.ring

`security_sentinel_narrated` 33 assertions
base/test/system/security_sentinel_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les sections honnêtes propres du dossier, chaque perte à côté de son gain.

En performance. Une base de données de séries temporelles, un langage de requête et des tableaux de bord sont ceux du terrain aujourd'hui, et absents ici. Par conception, le volet exporte. Sa réponse d'échantillonnage est faite d'anneaux bornés qui abandonnent le plus ancien, et l'échantillonnage en tête ou en queue, c'est ce que le terrain fait mieux aujourd'hui. Le contexte distribué atteint HTTP aujourd'hui ; aller au-delà est l'étape suivante. L'écosystème d'intégrations et d'auto-instrumentation pour chaque cadre est celui du terrain. Le profileur est coopératif : il voit ce que vous encadrez, là où un profileur natif voit tout. Le gain qu'il revendique est que le jugement et l'action vivent à l'intérieur du programme : des budgets qui font échouer la construction, un réglage qui passe la gouvernance, une surveillance qui se facture elle-même.

En sécurité. Le registre est un anneau borné, et borné veut dire oublier. La télémétrie de l'hôte, du noyau et du réseau est absente ; le volet lit ce que les propres jointures du programme émettent. Les détections sont déclarées et non apprises ; les apprendre est l'étape suivante, et une détection de taux, le 30 septembre 2026, en couvre déjà une partie. La propre comparaison du dossier est qu'un corpus de règles au format du terrain est plus grand aujourd'hui que les huit détections livrées, et le terrain mène ici. Le flux de travail d'un centre d'opérations de sécurité est celui du terrain aujourd'hui. Le plus grand écart, dans les mots du dossier, est qu'une détection ne peut voir que ce qu'une jointure émet. Le gain est que l'événement est typé à la naissance, là où la porte refuse, de sorte qu'aucun analyseur ne se tient entre ce que le programme savait et ce que le détecteur juge.

En promesses. Les doctests du terrain vivent dans la chaîne d'outils de l'hôte et répondent réussite ou échec ; c'est de la connaissance générale. Le juge de la maison a cinq états. Là où le terrain mène, c'est en instruments : la note de lancement de Testoor, le volet qui tient le voyageur, compte les instruments de couverture de toute sorte à zéro aujourd'hui, et la couverture de routes est l'étape suivante.

En perception. Les services d'instantanés visuels du terrain sont mûrs et automatiques ; c'est de la connaissance générale. La loi de la maison dit qu'une perception ne remplace pas un nombre, et qu'aucun nombre ne la remplace.

L'étude du dossier donne le ton que j'essaie de faire résonner. Une grille de deux cent quatre-vingt-six coches contre deux cent soixante-neuf croix est, dans ses mots, la forme d'une brochure.

Voyons maintenant où le chapitre s'arrête.

7. LA LIMITE

CONSTRUIT	la promesse jugée comme une affirmation sur un arrêt ; le volet de sécurité avec son registre, sa sentinelle, son exercice, sa réponse et ses secrets, et sa table de garanties liées à des gardes ; le volet de performance avec son instrument et sa boucle gouvernée ; la loi de la perception, adoptée
CONÇU	les routes, l'horloge et la météo, et la perception pour le lanceur de promesses ; le volet des extensions ; la négociation autour d'un plan rejeté
ABSENT	un verdict humain sur une image de la galerie ; sept des huit traditions musicales entendues par une personne qui en vient ; un vrai coffre et un vrai bac à sable comme preuves ; un magasin de séries temporelles

Ce qui est construit : les promesses comme arrêts ; le volet de sécurité, avec son registre, sa sentinelle, son exercice, sa réponse, son attestation et ses secrets ; le volet de performance de sa première phase à sa dernière ; la loi de la perception, adoptée. Et les tables de verdict déclarées pour la voix chantée et les huit univers. La table de garanties tient 43 lignes qui nomment 57 fichiers de garde distincts, aucun ne manquant.

Ce qui est conçu :

- Les routes et la carte du lanceur de promesses, son horloge et sa météo, ses excursions, sa perception et sa mutation, et ses tournées héritées. Tout cela est planifié. L'état *non perçu* est nommé dans l'en-tête du lanceur, et le lecteur n'en assigne aucun.
- La négociation autour d'un plan rejeté, que le modèle de menace appelle conception seulement.

- Le volet des extensions : les extensions ne sont pas livrées, et le code étranger n'est pas encore gouverné.

Ce qui est absent :

- **Un verdict humain sur une image de la galerie.** Zéro sur quatre-vingt-quinze. La deuxième exigence de la loi, un nom, une date et ce qu'ils ont dit, n'y est pas satisfaite. Le chapitre dit non perçu, et ne dit pas vérifié.
- Sur les huit traditions musicales, sept non entendues.
- Un vrai coffre et un vrai bac à sable comme preuves. Le résolveur de coffre est testé contre un remplaçant, sur un vrai HTTP. L'adaptateur de gestion de clés dans le nuage est absent.
- Les chemins sur les systèmes POSIX, non vérifiés, en attendant une machine Linux ; le fuzzing planifié ; l'intégration continue ; les commits signés, que j'ai différés pour la phase de développement.
- Un magasin de séries temporelles. Un pourcentage de couverture, qui est refusé tant qu'il n'y a pas de routes.

Trois autres choses ont leur place ici, trouvées en exécutant le tribunal.

Les nombres de l'exercice diffèrent selon la source. La mesure de réactivité du modèle de menace vient de trois exécutions. C'est une détection en un peu plus de trois cent cinquante millisecondes, et un confinement en un peu plus de cinquante. Le site imprime 395 et 50, d'un rejeu du 1er octobre. Ils diffèrent parce que l'exercice mesure une horloge. Une forme d'attaque, en boucle locale, sur une seule machine, lancée à la demande. Quel que soit le nombre que vous imprimez, vous nommez sa source.

Un défaut possible dans le juge de promesses, que je signale et sur lequel je n'ai rien construit. Un fichier dont la première promesse est un mensonge et la dernière une levée d'erreur promise lit le mensonge comme non atteint et la levée d'erreur comme tenue. La tournée est brisée, avec zéro divergente, et le code de sortie est 1. Je l'ai reproduit deux fois. Chaque règle seule se comporte comme la

garde la plante. Savoir si une divergence au-dessus d'une levée d'erreur promise devrait se lire *divergente* est une question pour le volet Testoor, qui tient le voyageur.

Des comptes qui ne s'accordent pas entre les sources. La clôture de l'étude dit 387 gardes narrées. Mon `find` sur l'instantané épinglé dit 529 fichiers ainsi nommés. Le compte propre de la construction est celui qu'il faut utiliser, et aucun des deux autres.

Six portes suivent, une pour chaque lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vos exemples portent déjà leur sortie attendue. Le voyageur les lit comme des promesses et dit tenue, divergente ou non jugée, et la prose qui se dispute avec elle-même n'est jamais accusée.
L'analyste	Le modèle de menace est une table que vous pouvez lire : chaque ligne avec la garde qui la prouve et sa limite honnête. Le jugement de performance est déclaré avant que le code ne s'exécute.
Le concepteur	Une image ou un son est une affirmation que seule une personne peut clore. La galerie dit qu'un modèle d'IA a regardé et qu'aucun humain ne l'a fait. Votre verdict est une ligne nommée dans le dossier.
Le décideur	La question de savoir qui a décidé, et ce qui est écrit, a une réponse mécanique dans les deux volets. La même action est refusée pour le modèle d'IA et validée pour l'humain, les deux dans un seul audit, et la preuve est scellée de sorte qu'une modification après la première entrée soit nommée.
Le formateur	La règle du tuteur rencontre le tribunal. Prédisez le verdict d'une promesse avant que le voyageur ne la parcoure.
L'agent	Vous pouvez proposer un plan de confinement ou un plan de réglage et vous ne pouvez valider ni l'un ni l'autre. Votre refus est lui-même un événement enregistré. Ne dites pas à une personne qu'une image est juste. Dites « non perçu ».

9. EXERCICE

Prédisez, par écrit, avant d'exécuter quoi que ce soit. Le fichier ci-dessous contient trois promesses. Quel état aura chacune ?

HARO

```
write("mine.haro", '? 2 * 3' + nl + '#--> 6' + nl +
        '? 2 * 3' + nl + '#--> 7' + nl +
        '? "ok"  #--> "ok" but see why' + nl)
# before you run: which state will each of the three promises have?
oMine = TravellerQ().Silent().Walk("mine.haro")
aMine = oMine.Records()[1][:run][:stops][1][:claims]
# your three predictions, then:  ? aMine[1][:state]   ? aMine[2][:state]   ? aMine[3][:state]
remove("mine.haro")
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Les états sont tenue, divergente et non jugée. Puis ajoutez une quatrième ligne qui lève une erreur, promettez-la avec `#--> ERROR:`, et prédisez d'abord comment la tournée finira. Ne changez pas votre prédiction pour la faire correspondre à ce que vous voyez. Expliquez d'abord la sortie.

Je vous laisse celui-ci. Voici le bloc de la levée d'erreur de la section 3 avec un seul changement : sa première promesse est maintenant un mensonge.

HARO

```
write("raise.haro", '? "before"  #--> after' + nl +
        '? Boom()  #--> ERROR: boom happened' + nl +
        'func Boom()' + nl + char(9) + 'raise("boom happened")' + nl)
oRaise = TravellerQ().Silent().Walk("raise.haro")
aRun = oRaise.Records()[1][:run]
# what does the tour say of each promise, how does it end, what is the exit code?
? aRun[:stops][1][:claims][1][:state]
? aRun[:stops][1][:claims][2][:state]
? aRun[:end]
? oRaise.ExitCode()
remove("raise.haro")
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Lisez ce que le voyageur dit du mensonge, ce qu'il dit de la levée d'erreur, comment la tournée finit et quel est le code de sortie, puis comparez avec ce que chacune des deux promesses a donné seule à la section 3. La section 7 dit pourquoi je ne peux pas vous le dire : c'est un défaut possible du juge de promesses, que j'ai signalé sans rien construire dessus, et que j'ai laissé non corrigé exprès. Quand vous saurez dire ce que le tribunal répond ici, vous connaîtrez la question que la section 7 confie au volet Testoor.

Le chapitre 22 est *Comparé, honnêtement*. Son principe est celui-ci : une comparaison ne vaut rien tant que la version de l'autre côté et la date ne sont pas sur la page, que l'autre côté n'a pas été exécuté, et que la perte n'est pas imprimée à côté du gain.

OÙ SONT LES PREUVES

NARRATION	`stz-security-ledger-narration.md`
NARRATION	`stz-governance-lineage-narration.md`
NARRATION	`stz-perf-governed-loop-narration.md`
NARRATION	`stz-honest-stopwatch-narration.md`
NARRATION	`stz-security-drill-narration.md`
SUR LE SITE	security
SUR LE SITE	offering
DÉCISIONS DE L'ÉTUDE	12.13, 12.16, 12.19, 12.20, 12.22, 12.26, 12.29

PARTIE IV · CHAPITRE 22

Comparé, honnêtement

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 5 bloc(s) sur 5 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je compare Softanza au terrain, et je dis comment une comparaison doit être faite avant de pouvoir être imprimée. Je montre cinq paires, la même tâche exécutée des deux côtés, avec la version de l'autre côté et la date sur la page et une ligne où l'autre côté mène. Je dis ce que j'ai refusé, la grille seule d'abord, et quelles gardes prouvent les blocs, puisque le tribunal des comparaisons est proposé et non construit. J'imprime la liste de ce que le terrain a et de ce qui manque à Softanza, à côté des gains. Le principe est celui-ci. Une comparaison ne vaut rien tant que la version de l'autre côté et la date ne sont pas sur la page, que l'autre côté n'a pas été exécuté, et que la perte n'est pas imprimée à côté du gain.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

La comparaison, sur le terrain, est un genre du marketing avec quelques exceptions honnêtes. Le dossier nomme le genre : une page « Compare » pour chaque rival, et un bloc de comparaison sur chaque page de fonctionnalité. Sa forme la plus courante est la grille, avec des lignes de fonctionnalités, des colonnes d'outils, des coches et des croix.

Les exceptions honnêtes sont des pratiques et non des produits. Des bancs d'essai publiés avec la méthode, la machine et le mode de construction. Et des sections qui disent ce que l'autre outil a et qui nous manque. C'est de la connaissance générale, et non une étude du dossier, et je ne nomme aucun outil.

2. LE PREMIER PRINCIPE

Une comparaison ne vaut rien tant que la version de l'autre côté et la date ne sont pas sur la page, que l'autre côté n'a pas été exécuté, et que la perte n'est pas imprimée à côté du gain.

Elle est faite de trois phrases du dossier. Mon étude du dossier, du 7 octobre 2026 : toute affirmation sur un autre outil est citée, datée et exécutée, et toute perte est publiée avec les gains. L'étude du domaine de projets sur un runtime de navigateur, exécutée le 8 août 2026 : les pertes sont publiées avec les gains. Et la page de couverture du site : une plateforme qui montre ses écarts est crue sur ses forces. Ma demande, telle que l'étude la rapporte, est un ton comparatif qui est « a motor force, to learn from the best and beat them when possible ». L'étude ajoute : avec « when possible » mesuré.

3. LA FORME SOFTANZA

Le chapitre 3 a dit « J'ai refusé la grille seule ». Ce chapitre est la méthode qui la remplace, avec ses preuves. Qu'a mesuré le dossier sur la grille ? L'étude a parcouru le texte de la bibliothèque à la recherche de tables dont l'en-tête nomme Softanza et au moins un autre outil. Elle a trouvé soixante-quatre tables, six cent quinze lignes, dans trente et un documents. Dans celles-ci, la colonne de Softanza porte 286 coches, 2 avertissements et 2 croix. Les autres colonnes portent 204 coches, 207 avertissements et 269 croix. Le jugement de l'étude est que c'est la forme d'une brochure, et que le dossier lui-même ne s'y fie pas. Je n'ai pas pu reproduire ces chiffres : un parcours plus étroit de ma main, avec une règle d'en-tête que j'ai écrite, a trouvé trente-sept tables dans vingt-neuf documents. L'étude n'imprime pas sa règle d'en-tête. Je cite ses nombres comme les siens, et datés.

La forme qui remplace la grille est une paire : la même tâche, les deux côtés exécutés, la version de l'autre côté sur la page, et une ligne où l'autre côté mène. J'en montre cinq. Quatre ont été exécutées des deux côtés, et la cinquième, le bloc cinq, du côté Softanza seulement. Les exécutions datent du 8 octobre 2026, entre 22 h 24 et 22 h 34, sur une machine Windows, et la construction du livre a ré-exécuté les

blocs de Softanza le 9 octobre 2026 à 05 h 32. Tous tenus. L'autre côté est Python 3.13.15, et Node v22.20.0 avec son moteur V8 12.4 et ICU 77.1. R et Julia ne sont pas installés sur cette machine, et aucune paire ne s'en sert.

Une : un entier est exact à toute taille. Combien de chiffres garde le produit de deux nombres de vingt ? Exécutez-le :

HARO

```
oN = new stzNumber("12345678901234567890")
? oN.MultiplyByQ("98765432109876543210").Content()
#--> 1219326311370217952237463801111263526900
```

✓ exécuté : a affiché ce que ce bloc promet

Quarante chiffres, exacts ! L'autre côté, Python 3.13.15, imprime les mêmes chiffres, `1219326311370217952237463801111263526900`, pour `12345678901234567890 * 98765432109876543210`. L'entier de Python est exact par défaut, et Softanza est à égalité avec lui. La ligne de Softanza est la plus longue : un littéral aussi long ne peut pas être un nombre natif, donc il est donné comme chaîne. L'autre côté, Node v22.20.0, imprime `1.219326311370218e+39`, parce que son nombre par défaut est un flottant de 64 bits. Il n'est exact que lorsqu'on le demande avec le suffixe `n`, un second genre de nombre. Donc où est le gain ? Sur le défaut de JavaScript, et non sur Python.

Deux : des décimaux exacts. `0.1` plus `0.2` fait-il `0.3` ici, et un prix multiplié par un taux garde-t-il ses centimes ? Exécutez-le :

HARO

```
? (new stzNumber("0.1")).AddQ("0.2").Content()
#--> 0.3
? (new stzNumber("19.99")).MultiplyByQ("0.15").Content()
#--> 2.9985
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `0.3` et `2.9985` : les centimes sont gardés ! L'autre côté, Python avec des flottants, imprime `0.30000000000000004` et `2.9984999999999995`. Node imprime les deux mêmes, parce que JavaScript n'a pas de type décimal. Le module standard `decimal` de Python, importé et nourri de chaînes, imprime `0.3` et `2.9985` : à égalité avec Softanza, et au même coût par ligne. Donc Softanza gagne sur le défaut, puisqu'un nombre fait à partir d'une chaîne est un décimal exact, et est à égalité avec l'unique bibliothèque qui le fait délibérément. Le commentaire de la garde dit pourquoi : la valeur est stockée comme chaîne, et le calcul est fait sur des entiers mis à l'échelle par le moteur. Sa limite est dans les mots du dossier, et l'un d'eux est périmé. Les échelons pour les rationnels et pour les nombres complexes sont nommés dans le plan et, dans un commentaire de la garde des nombres qui ne sont pas perdus, « not built yet ». L'instantané contient une garde narrée pour chacun, `numeric_rationals_narrated` et `numeric_complex_narrated`, de sorte que ce commentaire est périmé et que je ne m'y fie pas. Je n'ai pas exécuté ces deux gardes.

Trois : trier des mots accentués. Ici le terrain mène. Softanza trie par point de code. Où cela place-t-il `étoile` ? Exécutez-le :

HARO

```
? @@( Q([ "zèbre", "étoile", "eau", "zoo" ]).SortedInAscending() )
#--> [ "eau", "zoo", "zèbre", "étoile" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `[ "eau", "zoo", "zèbre", "étoile" ]` : `étoile` en dernier, après `zoo` ! Le tri du terrain, conscient de la locale, place `étoile` entre `eau` et `zèbre`. L'autre côté, le `sorted` de Python, imprime la même liste : à égalité, et tous deux faux pour le français. Avec le module standard `locale` réglé sur `fr_FR.UTF-8`, trié avec `locale.strxfrm`, Python imprime `eau, étoile, zèbre, zoo`. Cela a fonctionné sur la machine de l'exécution. La locale doit exister sur l'hôte, et l'appel vaut pour tout le processus. Node, avec `Intl.Collator('fr')`, imprime le même ordre français, `eau, étoile, zèbre, zoo`, avec une bibliothèque intégrée. Donc le terrain mène : JavaScript avec un argument, Python avec la locale de l'hôte. Softanza n'a pas de collation. Le plan de culture du 18 août 2026 dit que le tri est le tri ordinal

du moteur. Et la page de couverture note l'internationalisation partielle, contre profonde pour deux autres plateformes, avec une note : un pari délibéré contre la grande bibliothèque internationale.

Ce bloc est un écart épinglé. Sa promesse enregistre la réponse actuelle de la bibliothèque, de sorte que la page échoue le jour où la bibliothèque gagne la collation. C'est voulu. La règle propre du plan de culture dit qu'épingler un écart connu n'est légitime que lorsque l'assertion échoue le jour où l'écart est comblé, et lorsque l'étiquette le dit.

Quatre : la majuscule turque. Le terrain mène. Que répond la méthode quand on lui donne la locale turque ? Exécutez-le :

HARO

```
? Q("istanbul").UppercasedInLocale("tr_TR")
#--> ISTANBUL
```

✓ exécuté : a affiché ce que ce bloc promet

Elle répond `ISTANBUL` : la locale a été acceptée et ignorée ! Python imprime `ISTANBUL` : à égalité, puisque la bibliothèque standard n'a pas de casse selon la locale. Node, avec `toLocaleUpperCase('tr')`, imprime `İSTANBUL`. Le turc a besoin de la majuscule pointée. C'est la perte la plus digne d'être imprimée, et ce n'est pas que le moteur manque de turc. C'est que la méthode prend une locale et, aujourd'hui, répond en majuscule simple sans le dire ; honorer la locale, ou refuser, est l'étape suivante. Le commentaire de la source dit : majuscule simple quand elle n'a pas de règle. Le plan de culture a mesuré la même chose le 18 août 2026 et l'a appelée l'équivalent en code de la fausse barre de progression. Sa loi est la raison de ce chapitre : pas de façades. Une opération honore la culture qu'on lui a donnée, ou elle refuse. Je tiens cela pour une vue, non pour un résultat, et la loi du plan de culture dit la même chose.

Cinq : une instruction dans votre langue. Une instruction en langage naturel est guidée par un dictionnaire et locale : pas de réseau, pas de modèle d'IA, pas de clé. Que fait-elle de la phrase, et qu'a-t-elle compris ? Exécutez-le :

HARO

```
o1 = Naturally("Make a list holding [ 3, 1, 3 ] then drop its duplicates")
? @@( o1.Result() )
#--> [ 3, 1 ]
? o1.Understood()
#--> create a list with [ 3, 1, 3 ] -> remove duplicates
```

✓ exécuté : a affiché ce que ce bloc promet

Elle répond `[ 3, 1 ]`, et dit ce qu'elle a compris : `create a list with [ 3, 1, 3 ] -> remove duplicates`. Je n'ai pas exécuté l'autre côté. L'équivalent du terrain de « prends cette phrase et fais-le » est un appel de modèle d'IA. Ce qu'il lui faut, c'est un modèle. Un modèle hébergé demande un réseau, un compte, une clé et un coût par jeton ; un modèle local demande un modèle téléchargé et un GPU ou un processeur patient. Il lui faut une invite, et une vérification que la réponse est ce qui a été demandé, et la réponse n'est pas garantie d'être la même deux fois. La propre paire du site de ce genre écrit l'autre côté comme un commentaire, qui dit qu'il n'y a pas de pendant sans un modèle de langage externe.

Une perte de plus, qui n'est pas un bloc. Une chaîne d'un emoji de famille, cinq points de code, reliés par des liants, montre où la chaîne publique n'expose pas encore ce que le moteur sait. La classe compteur répond `1` pour le comptage de graphèmes ; `NumberOfChars` répond `5` et `NumberOfBytes` répond `18` ; et `GraphemeCount()` sur la chaîne elle-même lève une erreur, parce que la méthode n'est pas définie. Le moteur sait, et la chaîne ne le dit pas. Le `len` de Python donne `5`, et le `.length` de Node donne `8`.

Voilà la forme, cinq paires et une perte de plus ; ce que j'ai refusé pour y arriver vient ensuite.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé la grille seule. Deux cent quatre-vingt-six coches dans la colonne de Softanza contre deux cent soixante-neuf croix dans les autres colonnes, c'est la forme d'une brochure.

J'ai refusé de garder un chiffre phare qui ne se reproduit pas. La note du domaine de projets du 13 septembre 2026 a remesuré le runtime sous la bibliothèque contre un runtime rapide du terrain sur six noyaux appariés. Appariés veut dire algorithme et tailles identiques des deux côtés, chaque côté dans son idiome rapide, sommes de contrôle inter-langages affirmées avant tout nombre, minima sur des exécutions répétées. Le terrain mène sur chaque ligne, d'un facteur entre 1,04 et 1,88. La note dit que l'affirmation à retirer est que le runtime le bat, au présent, sur x64. Le chiffre phare n'a jamais été remesuré après que le prototype, que la note appelle « spike », est devenu un produit. La note m'a demandé de corriger les notes propres du runtime. Le chiffre phare a été retiré dans la note du domaine de projets du 13 septembre 2026, et je n'en dis pas plus.

J'ai refusé une rétractation qui supprime ses preuves. Les notes propres du runtime du 29 août 2026 ont rétracté une table publiée dans l'autre sens, une perte sur un téléphone. L'exécution avait eu lieu sur les cœurs lents du téléphone alors que le pair tournait sur des cœurs rapides. Les notes gardent la table, barrée, et disent qu'une rétractation qui supprime ses propres preuves n'est pas une rétractation. Donc le dossier a retiré un gain et retiré une perte. La règle est de reproduire avant d'imprimer. Imprimer des pertes n'est pas la règle. Les deux retraits que garde le dossier, un gain et une perte, sont ce qui la soutient.

J'ai refusé une comparaison qui ne peut pas être refaite. Le paquet de passage de relais du même jour, le 13 septembre 2026, dit que les noyaux du prototype sont morts avec l'arbre de travail. C'est pourquoi une session extérieure a dû reconstruire la mesure à partir de prose.

J'ai refusé une façade. Une opération honore la culture qu'on lui a donnée, ou elle refuse ; les blocs trois et quatre impriment où en est la bibliothèque là-dessus aujourd'hui.

J'ai refusé la grande bibliothèque internationale. La table du plan de culture dit qu'elle est tuée, une décision prise et en vigueur dans la première couche. Le coût, ce sont les blocs trois et quatre. La page de couverture l'imprime comme un pari délibéré.

J'ai refusé de comparer sur le terrain le plus fort de l'outil. L'étude du domaine de projets sur un runtime de navigateur a trois règles d'honnêteté. Chaque langage reçoit du code idiomatique ; les pertes sont publiées avec les gains ; chaque moteur est une construction de production. Elle dit, de sa propre page, que sans elles c'est du marketing. Son tableau de score a neuf scénarios, quatre gagnés et cinq perdus. Ce n'est pas une barrière de régression : ses nombres servent à apprendre, et non à barrer.

Le tribunal qui tiendrait ces refus vient ensuite, et il est proposé, non construit.

5. LE TRIBUNAL

Le tribunal des comparaisons est proposé et non construit. Le plan de l'étude le nomme : une ligne de comparaison sans source ou sans date ne se publie pas. Une paire de code dont l'autre côté n'a pas été exécuté, ou l'a été dans une construction de débogage, ne se publie pas non plus. Je n'ai trouvé aucun fichier dans la bibliothèque ou le site qui l'implémente. Donc qu'existe-t-il ? Quatre gardes, qui prouvent les blocs et touchent à la méthode.

`numeric_exactness_narrated` prouve l'arithmétique décimale exacte, les entiers au-delà de 2 puissance 53, le produit de quarante chiffres, et la division exacte quand elle se termine. `numeric_no_loss_narrated` prouve qu'un nombre n'est pas détruit en entrant, et que l'échelle de représentation est observable. Son commentaire final nomme encore comme non construits les échelons des rationnels et des nombres complexes, et il est périmé (voir le bloc deux). `semantic_resolver_narrated`

prouve que les paraphrases se résolvent en opérations canoniques par le lexique local, et la phrase du bloc cinq est l'un de ses cas. Elle ne passe pas à ce commit : je l'ai exécutée le 8 octobre 2026 et elle s'arrête sur une erreur levée par une fonction de découpage de chaîne, avant sa fin. Donc la page compte ses assertions et ne prétend pas à son verdict. `engine_is_optimised_narrated` prouve que le moteur est compilé avec optimisation, par une barrière de temps : c'est la garde propre de la bibliothèque pour la règle de la construction de production.

Aucune garde n'affirme les blocs trois et quatre comme des écarts épinglés. Aucune garde n'existe pour une paire de code dont l'autre côté n'a pas été exécuté. Les deux sont des demandes à la bibliothèque.

Le tribunal n'est pas construit, mais le dossier tient déjà la liste de ce que le terrain a, et elle suit.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`numeric_exactness_narrated` 39 assertions
base/test/number/numeric_exactness_narrated.ring

`numeric_no_loss_narrated` 31 assertions
base/test/number/numeric_no_loss_narrated.ring

`semantic_resolver_narrated` 83 assertions
base/test/natural/semantic_resolver_narrated.ring

`engine_is_optimised_narrated` 7 assertions
base/test/number/engine_is_optimised_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le dossier tient déjà la liste de ce que le terrain a et de ce qui manque à Softanza. Je l'imprime, par fichier et par date, à côté des gains.

La performance, face au terrain, 29 juillet 2026. Pas de magasin de séries temporelles, pas de langage de requête, pas de tableaux de bord : par conception, le volet exporte. Pas de contexte distribué au-delà de HTTP. Un échantillonnage honnête, là où celui du terrain est plus fin aujourd'hui. Des décennies d'intégrations, que les titulaires possèdent. Les mots propres de l'étude sont que sur l'étendue les titulaires gagnent décisivement, et que le volet ne les concurrence pas et ne devrait pas feindre de le faire. Quatre pertes sont barrées avec la phase qui les a fermées.

L'analyse d'incident, face au terrain, 1er août 2026. L'échelle, la rétention et la recherche, là où borné veut dire oublier. Pas de télémétrie de l'hôte, du noyau ou du réseau. Un petit ensemble de détections livrées face au corpus d'une communauté. Le flux de travail d'un centre d'opérations. Et les jointures non câblées, que l'étude appelle le plus grand écart honnête du système, listées et non cachées.

L'étude des extensions, 16 août 2026. Le terrain est en avance sur la granularité des permissions, sur la résolution des dépendances, sur la distribution et sur la signature. L'en-tête de l'étude dit que c'est de l'analyse et non des règles.

Le plan de culture, 18 août 2026. Chaque ligne a été vérifiée dans cette session en l'exécutant. La casse prend une locale sur quatre niveaux et ne l'honore pas encore. La collation, la mise en forme des chiffres, les calendriers non grégoriens et les fuseaux horaires sont absents. Le statut du plan était alors non commencé.

L'étude du domaine de projets sur le runtime de navigateur, 8 août 2026. Quatre lignes sur neuf gagnées, cinq pour les pairs, et l'opcode de boucle fusionné d'un pair reste en avance d'une génération de conception.

Le runtime face à un pair rapide, 13 septembre 2026. Six noyaux, les sommes de contrôle d'abord, le pair en avance sur chaque ligne.

Le banc d'essai de chaînes du 7 juillet 2026, et la bataille polyglotte, dans une narration datée du 18 avril 2025 : l'ancien langage hôte est arrivé dernier sur sept dans deux manches sur trois. Il est arrivé dernier au total aussi. Les deux portent des réserves que le dossier ultérieur y a mises. Je n'ai pas ré-exécuté le banc d'essai. La construction par défaut du moteur était une construction de débogage jusqu'au 26 juillet 2026, et le fichier de construction propre de la bibliothèque dit que toute bibliothèque partagée qu'elle avait jamais chargée était compilée sans optimisation. Donc le banc d'essai a probablement tourné sans optimisation. C'est une inférence, et je l'énonce comme telle. Le même commentaire dit que, sur le travail de chaînes, de listes et de matrices, ce changement n'achète qu'environ quinze pour cent. Donc le mode de construction est une réserve sur les nombres et non une excuse pour eux. Je n'ai pas pu lire le mode de construction du moteur sur lequel ce livre tourne, et l'horloge de l'ancien langage hôte résout dix à quinze millisecondes ; une ré-exécution équitable exige de lire les deux. Le banc d'essai est donc cité ici comme un dossier daté, avec cette réserve, et non comme une mesure présente.

La matrice de couverture. Trente domaines en sept groupes, face à quatre autres plateformes, lue le 17 septembre 2026. Le bilan de Softanza est sept profonds, quatorze solides, sept partiels, deux absents, présent dans vingt-huit sur trente. Je l'ai recompté à partir des lignes et il correspond. Les deux lignes absentes sont les mathématiques symboliques (« no computer algebra ») et le paquet, la construction et le déploiement (« a library, not a registry »). Parmi les sept lignes partielles se trouve l'internationalisation, « a deliberate bet against ICU », et la ligne neuronale est « inference only; no training foundry ». Les bilans stockés de deux des autres plateformes ne correspondent pas à leurs propres lignes, d'un en deux endroits. Je n'imprime que le bilan de Softanza. La matrice cite un fichier boussole comme source des notes. Je n'ai pas pu localiser le fichier dans l'instantané, dans la copie de travail ni dans le dépôt du domaine de projets.

La comparaison de graphes, une narration datée du 26 décembre 2025. Un modèle d'IA l'a écrite, et le dit dans sa signature. Sa colonne pour Softanza se note elle-même d'une croix sur le passage à l'échelle et sur les transactions et d'un avertissement sur la performance, et sa conclusion énumère où les solutions traditionnelles gagnent.

Là où Softanza se distingue, selon le dossier, sans un nombre : le SLA ; l'acte gouverné ; l'événement typé ; la seule ligne de la matrice sur la gouvernance et l'exécution sûre. Deux de plus : des nombres exacts par défaut, face à deux flottants ; une instruction qui se résout localement. Et une méthode qui imprime chacun d'eux à côté d'une perte.

Voilà la liste ; je dis maintenant où ce chapitre s'arrête.

7. LA LIMITE

CONSTRUIT — cinq comparaisons, quatre exécutées des deux côtés avec la version de l'autre côté et la date, et la cinquième exécutée du côté Softanza seulement ; les quatre paires de code du site, ré-exécutées, trois d'entre elles des deux côtés ; trois études de terrain qui finissent sur ce que le terrain a et que la bibliothèque n'a pas, et un plan de culture qui imprime ses pertes

CONÇU — un registre de comparaisons comme données ; un tribunal des comparaisons dont la règle est qu'une ligne sans source ni date ne se publie pas ; les deux bancs d'essai ré-exécutés sur le moteur actuel

ABSENT — la collation selon la locale ; la casse selon la locale ; un comptage de graphèmes sur la chaîne publique ; une base de fuseaux horaires ; le banc d'essai du 7 juillet 2026 ré-exécuté ; la source de la matrice de couverture, que je n'ai pas pu localiser

Ce qui est construit : quatre des cinq paires, exécutées des deux côtés, et la cinquième du côté Softanza. Les quatre paires de code du site sont construites aussi. Je les ai ré-exécutées le 8 octobre 2026, trois d'entre elles des deux côtés (la quatrième a un commentaire pour son autre côté). Les sorties concordaient. Le sont aussi les études de terrain de la bibliothèque sur la performance, l'analyse d'incident et les extensions, avec leurs sections de pertes, et le plan de culture, qui imprime ses pertes. Le sont aussi le harnais du banc d'essai de chaînes et la garde que le moteur est optimisé.

Ce qui est conçu ou proposé : un registre de comparaisons comme données ; un tribunal des comparaisons ; la ré-exécution des deux bancs d'essai. L'étude tient les trois. Le 8 octobre 2026, je les ai cherchés dans la bibliothèque et dans le site et n'ai rien trouvé. Le site, lu le 9 octobre 2026, imprime cinq règles pour comparer, la première étant « Cite or do not publish ». Il imprime aussi une liste des neuf lignes où Softanza est partiel ou absent, avec les plateformes qui y sont profondes. Il dit, dans une ligne marquée conçu, qu'un registre de comparaisons n'est pas encore tenu comme données. Il ne nomme pas le tribunal.

Ce qui est absent : une colonne *comparé* sur l'Atlas, sur le registre des innovations et sur les narrations. Absent du site : les études de terrain, les soixante-quatre grilles elles-mêmes (il en donne maintenant le nombre, et rien de plus), les deux bancs d'essai et la table du runtime. Absent de la bibliothèque : la collation selon la locale ; la casse selon la locale ; un comptage de graphèmes sur la chaîne publique ; une base de fuseaux horaires. Et absent encore : la ré-exécution du banc d'essai du 7 juillet 2026, et la source de la matrice de couverture.

Et quelque chose qui n'est pas reproduit : les soixante-quatre tables, six cent quinze lignes et trente et un documents de l'étude, que mon parcours n'a pas reproduits, et les comptes de symboles.

La méthode est plus stricte que le dossier auquel elle est appliquée. Des comparaisons plus anciennes de la bibliothèque précèdent ses règles et ne satisfont pas encore l'une d'elles ou plus. Le banc d'essai de chaînes a probablement tourné sur un moteur de débogage. La bataille polyglotte n'a ni versions ni dates dans son texte, et dit que ses temps sont relatifs à un environnement informatique précis. La grille de graphes n'a ni versions, ni dates, ni sources pour ses cellules. La grille de charge cognitive d'un autre article de conception avance des chiffres de soixante à soixante-dix pour cent sans méthode imprimée à côté. Cet article finit ensuite sur un positionnement honnête qui nomme quatre choses sur lesquelles la bibliothèque ne concurrence pas.

Six portes suivent, une pour chaque lecteur de cette page.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Copiez une paire. Exécutez les deux côtés dans vos propres versions, gardez la sortie telle qu'elle est venue, et ajoutez la ligne où l'autre outil est en avance.
L'analyste	La matrice de couverture n'est une aide à l'achat que parce qu'elle montre deux lignes absentes et sept partielles dans ses propres couleurs. Lisez celles-là d'abord.
Le concepteur	La perte va dans le même bloc et dans la même typographie que le gain. Le mot d'étape fait partie de l'interface.
Le décideur	Demandez à tout fournisseur, celui-ci compris, la version de l'autre côté, la date, si cela a été exécuté, et la ligne qu'ils perdent. La note du 13 septembre 2026 est un modèle de chiffre phare retiré.
Le formateur	Une paire de code est une leçon sur ce que coûte une ligne. L'apprenant écrit la sortie de l'autre côté avant de l'exécuter.
L'agent	Quand vous répétez une affirmation sur un autre outil, portez sa version et sa date, et dites si vous ne l'avez pas exécutée. Le registre qui rendrait cela vérifiable par un agent d'IA est proposé et non construit.

9. EXERCICE

Choisissez une tâche où vous croyez que votre propre outil est en avance, et une où vous croyez que l'autre l'est. Faites une comparaison qui obéit aux trois règles : la version de l'autre côté et la date, les deux côtés exécutés, la perte imprimée.

HARO

```
? YourSide(the_input)
#--> the line your tool prints
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Sous elle, écrivez comme texte : le nom et la version de l'autre outil, la date, la ligne exacte qu'il imprime, et une ligne de la forme « ici l'autre mène parce que… ». Un lecteur vérifie les trois lignes de texte, puisqu'un lanceur ne le peut pas.

Une variante qu'une machine peut juger : prenez les trois exécutions de l'autre côté du bloc trois, le `sorted` de Python, Python avec la locale, et Node avec le collateur, et prédisez, avant d'exécuter, l'ordre que chacune imprime. Je ne vous dirai pas quel ordre est le bon.

Je vous laisse celui-ci. Voici le bloc trois, sa promesse retirée :

HARO

```
? @@( Q([ "zèbre", "étoile", "eau", "zoo" ]).SortedInAscending() )
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Sous lui, écrivez la promesse qu'attendrait un lecteur français, l'ordre que donne le tri conscient de la locale du terrain à la section 3, comme une ligne `#-->`, et exécutez-le. Je n'imprime pas le verdict. Lisez-le, puis lisez les sections 3 et 7 pour apprendre pourquoi ce bloc est un écart épinglé et non une faute du tri. Quand vous aurez résolu cela, vous saurez pourquoi le bloc trois promet la réponse actuelle de la bibliothèque et non celle du terrain.

Le chapitre 23 se tourne vers l'architecte, et vers un monde, une constellation et une plateforme. Là, la solution est déclarée en mondes, en liens et en une enveloppe avant que rien en soit construit, et chaque traversée entre eux est gouvernée.

OÙ SONT LES PREUVES

NARRATION	stzgraph-vs-other-graph-libraries.md
NARRATION	stzexcis-performance-battle.md
NARRATION	stz-vs-qt-unicode-narration.md
NARRATION	stznatural-vs-ring-naturallib.md
NARRATION	stz-near-natural-language-programming.md
SUR LE SITE	compare
SUR LE SITE	code
DÉCISIONS DE L'ÉTUDE	12.10, 12.22, 12.29

PARTIE V

La plateforme, assemblée

23 · Un monde, une constellation, une plateforme, et l'architecte
24 · Répéter, puis valider
25 · Vos utilisateurs gardent leur médium
26 · L'environnement
27 · Le moteur sous-jacent : trois couches, un moteur, Haro

PARTIE V · CHAPITRE 23

Un monde, une constellation, une plateforme, et l'architecte

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 10 bloc(s) sur 10 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je déclare une solution à trois niveaux, un monde, une constellation et une plateforme, et je nomme la seule personne qui déclare les trois, l'architecte. Je prends d'abord les trois habitudes du terrain : la super-appli, la carte de contextes et l'équipe de plateforme. Puis je fais tourner un monde qui s'explique lui-même, une constellation dont les traversées sont refusées une barrière à la fois, et un profil qu'une plateforme refuse de construire quand il n'est pas sain. Je dis où le code s'arrête : les vérités d'un monde sont narrées et non jugées, les coques sont minuscules, et la certification de l'architecte est une proposition. Le principe est celui-ci : la solution est déclarée en mondes, en liens et en une enveloppe avant que rien en soit construit, et chaque traversée entre eux est gouvernée.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Trois habitudes du terrain ont chacune un foyer, et je les prends de la connaissance générale, non d'une étude du dossier. Le dossier ne tient aucune étude de terrain sur les super-applis, sur l'ingénierie de plateforme ou sur les cartes de contextes. Son seul terrain nommé est une ligne d'un document de conception : une célèbre super-appli de messagerie, et ses semblables.

La première est la super-appli et le mini-programme. Une grande application embarque un runtime et fait tourner de petites applications en elle, partageant ses services et son magasin. L'hôte est privilégié, la mini-appli de seconde classe, et

l'isolation un bac à sable.

La deuxième est le contexte borné et la carte de contextes de la conception pilotée par le domaine, de 2003. Un grand système est découpé en modèles, chacun avec ses mots, et une carte montre comment ils se relient. C'est l'idée du terrain la plus proche d'un monde et d'un lien. Elle vit dans des diagrammes et en revue, et non dans un registre vivant qui refuse un appel.

La troisième est l'équipe de plateforme et le déploiement déclaré comme code. Une équipe construit la base interne sur laquelle les autres déploient, et des outils déclarent des machines avant qu'elles existent. La personne qui travaille à travers les étages est, sur le terrain, un architecte logiciel, un architecte de solutions ou un ingénieur de plateforme. C'est un titre tenu par une personne, dont les artefacts sont des diagrammes et des manifestes.

Pour les portées, le foyer du terrain est le compilateur : la cible d'une construction, et la vérification qui signale un appel que le niveau minimal de plateforme n'a pas. Chacun décide pour quel système le code est fait. Aucun ne laisse le programme nommer ce système au site d'appel et refuser l'acte.

La forme partagée : le terrain dessine l'image, un hôte, une carte, un diagramme, et garde le juge à l'extérieur. Dans les trois, le programme ne sait rien de sa propre composition. C'est ma lecture des trois, non une enquête sur le terrain.

À cette forme, j'oppose un principe.

2. LE PREMIER PRINCIPE

La solution est déclarée en mondes, en liens et en une enveloppe avant que rien en soit construit, et chaque traversée entre eux est gouvernée.

Les phrases du dossier sont trois. Une application est un monde vivant de sens. Des mondes, non des mini-applis : un graphe gouverné et non un conteneur hôte. Un architecte Softanza modélise d'abord l'architecture de déploiement, comme artefact déclaratif, avant une ligne de code de fonctionnalité système. La phrase ci-dessus est faite des trois et de la méthode du livre, qui est de déclarer puis de gouverner. Les trois sont celles du dossier, dites avec mes mots ; la phrase ci-dessus est le principe propre du chapitre, dans la voix du livre.

3. LA FORME SOFTANZA

Trois niveaux, donc. Un monde est une application. Une constellation est un graphe de mondes. Une plateforme est l'enveloppe autour d'une solution et le sol sur lequel se tient une constellation. Et une seule personne déclare les trois.

UN MONDE

Un monde a un être, un devenir, et une manière d'être rencontré. Le chapitre 4 a montré son être. Que répond un monde quand on lui demande d'expliquer les trois ? Demandez-le à un monde, pour une agence qui rend visite à des clients :

HARO

```
oApp = new stzApp("FieldVisits")
oApp {
        AddThingQ(:Account) { Has([ :number, :chapter, :balance ])  IsTrue(:balance, "@ >= 0") }
        AddThingQ(:Client)  { Has([ :code, :name, :city ])  Owns(:Account) }
        AddThingQ(:Visit)   { Of(:Client)  Has([ :agent, :date, :subject ]) }
        AddRelation(:Account, :belongsTo, :Client)
        AddFlowQ(:agent, :records, :Visit) { Require(:subject) }
        AddReactionQ(:Client).Unseen(90, :Days) { Propose(:Visit) }
        AddGoalQ(:EveryClientSeenThisQuarter) {
                Means      = "every :Client Has(:visit) Since(:quarterStart)"
                ReachedBy  = :planning
        }
        SetBody([ :GraphDB, :Files ]) { Graph = ".stzapp/world.stzgraf"  Files = "./" }
        AddScreenQ(:ClientFile) { ToUnderstand(:Client)  Shows([ :identity, :accounts, :visits ])
}
        AddRefinementQ(:balance).Bounds(0, 1000000)
        AddReaches([ :mobile, :desktop, :web ])
        Explain()
        #--> WORLD FieldVisits   lives in: graphdb + files
        #      BEING
        #        account (number, chapter, balance)
        #            true when balance @ >= 0
        #        client (code, name, city)
        #        visit (agent, date, subject)
        #      RELATIONS
        #        client owns account
        #        visit of client
        #        account belongsto client
        #      BECOMING
        #        when agent records visit require subject
        #        whenever client unseen 90 days -> propose visit
        #        wants everyclientseenthisquarter -> reached by planning
        #      MET FROM WITHOUT
        #        screen clientfile: understand client shows identity, accounts, visits
        #        refine balance bounds [0..1000000]
```

```
        #       reaches mobile, desktop, web
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il a répondu sous trois titres, `BEING`, `BECOMING` et `MET FROM WITHOUT`. `SetBody` ne fait que déclarer. Aucun fichier n'a été écrit par ce bloc.

Un monde tend vers quelque chose. Un but est un état voulu du graphe, et l'écart est mesuré. Un but sans motif à évaluer est refusé, et jamais simulé :

HARO

```
StzAppQ("visits") {
        AddThingQ(:Client) { Has([ :code, :name ]) }
        AddThingQ(:Visit)  { Of(:Client) }
        AddGoalQ(:EveryClientVisited) { Means = "every :Client Has(:visited)"  ReachedBy =
:planning }
        AddInstance("anna", :Client)
        AddInstance("bilal", :Client)
        Relate("anna", :visited, "visit1")
        ? @@( Pursue(:EveryClientVisited) )
        #--> pursuing everyclientvisited via planning -- 1 proposal(s)
        #    [ [ "attend", "client", "for", "bilal" ] ]
        ? GoalSatisfied(:EveryClientVisited)
        #--> 0
        Relate("bilal", :visited, "visit2")
        ? @@( Pursue(:EveryClientVisited) )
        #--> pursuing everyclientvisited via planning -- 0 proposal(s)
        #    [ ]
        ? GoalSatisfied(:EveryClientVisited)
        #--> 1
        AddGoalQ(:Vague) { Means = "make things nice" }
        try
                Pursue(:Vague)
        catch
                ? cCatchError
        done
        #--> stzGraphGoal.FromMeans: no 'every :Type' clause in 'make things nice'.
}
```

✓ exécuté : a affiché ce que ce bloc promet

`Pursue` imprime sa propre première ligne, puis le `?` imprime la liste qu'il renvoie. Une relation plus tard, l'écart est comblé : `0 proposal(s)`, et `GoalSatisfied` répond `1`. Le but vague est refusé avec sa raison : `no 'every :Type' clause in 'make things nice'`. Avec une réaction déclarée, le même écart propose une *visit* pour Bilal, et sans elle, comme dans l'exécution ci-dessus, l'acte par défaut est *attend* : `[ "attend", "client", "for", "bilal" ]`.

UNE CONSTELLATION

Rien ne passe entre les mondes à moins d'être déclaré et gouverné. Un lien doit exister, puis trois barrières se tiennent sur la traversée : l'acte doit avoir un niveau de risque déclaré, l'appelant doit détenir la permission, et l'autorité de l'appelant doit couvrir le risque. Chaque refus nomme sa barrière. Quelle barrière refuse la première ? Le lien existe dans les quatre appels ci-dessous, donc les trois refus sont ceux de la gouvernance :

HARO

```
oResto = StzAppQ("resto")
oResto.AddThingQ(:dish) { Has([ :name, :price ]) }
oSupplier = StzAppQ("supplier")
oSupplier.AddThingQ(:crate) { Has([ :produce, :qty ]) }
oCon = new stzSuperApp("acme-group")
oCon.AddWorld("resto", oResto)
oCon.AddWorld("supplier", oSupplier)
oCon.Bond("resto", "supplier", "order-produce")
? oCon.CallAcross("resto", "supplier", "order-produce")
#--> 0
? oCon.Why()
#--> governance refused: refused: 'order-produce' has NO declared risk tier (undeclared actions
never proceed)
oCon.GovDeclareRisk("order-produce", 2)
? oCon.CallAcross("resto", "supplier", "order-produce")
#--> 0
? oCon.Why()
#--> governance refused: refused: 'resto' lacks PERMISSION (can) for 'order-produce'
oCon.GovGrant("resto", "order-produce")
? oCon.CallAcross("resto", "supplier", "order-produce")
#--> 0
? oCon.Why()
#--> governance refused: refused: 'resto' holds '' authority (level 0) but 'order-produce' is
risk tier 2 (SHOULD does not cover it)
oCon.GovSetAuthority("resto", :Delegated)
? oCon.CallAcross("resto", "supplier", "order-produce")
#--> 1
```

```
? oCon.Why()
#--> allowed: active + bonded + governed
? oCon.GovernanceQ().NumberOfDecisions()
#--> 1
? @@( oCon.GovernanceQ().DecisionsOf("resto")[1][2] )
#--> [ "rationale", "cross-world call cleared" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le niveau de risque refuse le premier : `'order-produce' has NO declared risk tier`. Le mot doublé « refused: refused: » est le texte propre de la bibliothèque. La traversée autorisée a écrit une décision d'elle-même ! Son enregistrement tient l'autorité et le risque de cet instant, et une lecture d'horloge, de sorte que le bloc n'imprime que la justification : `cross-world call cleared`. C'est la filiation qu'utilise le chapitre 30.

Le sol partagé est un monde aussi, dans les mots de la conception : le monde zéro, uniforme et récursif, et non un genre particulier de nœud. Dans une constellation, le socle commun est lui-même une plateforme, et il tient une identité, une session et un magasin. De quelle classe est le socle commun dans le code ? Demandez-le-lui :

HARO

```
oCon = new stzSuperApp("acme-group")
oDb = new stzDatabase(":memory:")
oCon.OpenCommonsOn(oDb)
? classname( oCon.CommonsQ() )
#--> stzplatform
oCon {
        ? RegisterIdentity("owner", "s3cret")
        #--> 1
        ? RegisterIdentity("owner", "other")
        #--> 0
        ? len( OpenSession("owner", "s3cret") ) > 0
        #--> 1
        ? OpenSession("owner", "wrong") = ""
        #--> 1
        StorePut("currency", "EUR")
        ? StoreGet("currency")
        #--> EUR
}
oDb.Close()
```

✓ exécuté : a affiché ce que ce bloc promet

Ce bloc utilise une base de données en mémoire, donc ni fichier ni réseau. Le jeton de session est aléatoire, et seule sa présence est imprimée. Les secrets sont stockés salés et hachés, avec la fonction de dérivation de clé du moteur. La première ligne répond `stzplatform`, et c'est là le point : une plateforme n'est pas seulement l'enveloppe autour d'une solution. C'est ce sur quoi se tient une constellation. Dans le code, le socle commun n'est pas un nœud du graphe de la constellation : quand je l'ai sondé, le graphe tenait deux nœuds pour deux mondes enregistrés. Le monde zéro est un rôle que joue la plateforme.

Un monde échangé à chaud garde-t-il ses liens ? Une constellation s'emboîte aussi dans une autre, et un nom ne se prend qu'une fois :

HARO

```
oResto = StzAppQ("resto")
oResto.AddThingQ(:dish) { Has([ :name, :price ]) }
oCon = new stzSuperApp("acme-group")
oCon {
        AddWorld("resto", oResto)
        AddWorld("supplier", StzAppQ("supplier"))
        Bond("resto", "supplier", "order-produce")
        GovDeclareRisk("order-produce", 2)
        GovGrant("resto", "order-produce")
        GovSetAuthority("resto", :Delegated)
        oNew = StzAppQ("supplier-v2")
        Swap("supplier", oNew)
        ? NumberOfWorlds()
        #--> 2
        ? WorldQ("supplier").Name()
        #--> supplier-v2
        ? CallAcross("resto", "supplier", "order-produce")
        #--> 1
}
oHolding = new stzSuperApp("holding")
oHolding {
        AddConstellation("acme", oCon)
        ? KindOf("acme")
        #--> super
        ? WorldQ("acme").NumberOfWorlds()
        #--> 2
}
? oCon.KindOf("resto")
#--> app
try
    oCon.AddWorld("resto", oResto)
catch
    ? cCatchError
```

```
done
#--> stzSuperApp: world 'resto' already in the constellation.
```

✓ exécuté : a affiché ce que ce bloc promet

Le lien a survécu à l'échange : `CallAcross` répond encore `1` ! Le chapitre 28 utilise `Retire` et `Revive` ; ce bloc est `Swap` et `AddConstellation`. L'emboîtement est un fait du registre. Rien dans le code n'achemine un appel dans une constellation emboîtée, donc la récursion de la conception est celle d'un registre et pas encore celle d'un appel. Une traversée est un verdict : `CallAcross` autorise ou refuse par `1` ou `0`, et rien n'est encore remis à la cible.

Un piège. Un monde enregistré est une copie. L'ancien langage hôte copie un objet par valeur quand il est stocké, de sorte que le monde que vous enregistrez n'est pas le monde que vous tenez. L'en-tête de la classe le dit : les objets monde s'atteignent par l'index du registre, le chemin vivant. Configurez le monde entièrement, enregistrez-le, et atteignez-le par la constellation à partir de là. L'enveloppe a la même doctrine : poser un monde sur la plateforme récolte ses déclarations au moment de l'appel, et poser la gouvernance prend un instantané du régime.

L'ENVELOPPE, ET LA JOINTURE

Une plateforme n'accorde une capacité d'un appareil que si la gouvernance en autorise le risque. Une permission détenue suffit-elle ? Les niveaux de risque, deux pour la caméra et quatre pour les paiements, sont ceux que la bibliothèque a déclarés :

HARO

```
oGov = new stzGovernance("resto-regime")
oGov {
        GrantPermission("resto", "use-camera")
        GrantPermission("resto", "use-payments")
        SetAuthority("resto", :Delegated)
}
oResto = StzAppQ("resto")
oResto.AddThingQ(:dish) { Has([ :name, :price ]) }
StzPlatformQ("resto-envelope") {
        SetWorld(oResto)
        SetGovernedBy(oGov)
        AddCapabilityQ(:camera).SetPurposes([ :scan_dish_photo ])
```

```
        ? Granted(:camera)
        #--> 1
        AddCapabilityQ(:payments).SetPurposes([ :charge_guests ])
        ? Granted(:payments)
        #--> 0
        ? Why()
        #--> refused: 'resto' holds 'delegated' authority (level 2) but 'use-payments' is risk
tier 4 (SHOULD does not cover it)
}
```

✓ exécuté : a affiché ce que ce bloc promet

La permission était détenue pour les deux, et les paiements ont été refusés quand même ! La raison : `'resto' holds 'delegated' authority (level 2) but 'use-payments' is risk tier 4`.

L'en-tête de la classe dit ce qui manque au monde : le monde modélise le monde, et manque délibérément de l'enveloppe opérationnelle. Les devoirs de l'enveloppe sont la génération, la jointure des capacités, le runtime du socle commun, un corps en réseau et un registre avec application. La génération est réelle, et refuse de générer pour un monde qui ne déclare aucune atteinte. Le chemin du succès écrit de vrais fichiers, donc je ne l'exécute pas ici. Je l'ai exécuté une fois, pour un monde d'un écran et d'une chose, et j'ai retiré ce qu'il avait écrit. La coque web est une page de 318 octets avec les écrans et les choses du monde comme éléments de liste, et la coque de bureau un lanceur de 251 octets. D'autres mondes donnent d'autres tailles. Telle est la limite des « coques par plateforme » : la génération est réelle, et minuscule.

L'ARCHITECTE, ET LES TROIS PORTÉES

Par *portée* j'entends celle de la System Foundation : développement, déploiement et exécution courante. Chaque acte système est écrit dans une portée nommée. Un architecte Softanza modélise d'abord l'architecture de déploiement. Le même verbe, dans deux portées, reçoit deux verdicts, et les mots propres de la cible disent pourquoi :

HARO

```
oSol = new stzPlatformProfile("greenhouse")
oSol {
```

```
        DevelopedOn(:Windows)
        WithServer(:backend, :LinuxServer)
        WithSuperApp(:phone, :Android)
        WithApp(:sensor, :ESP32)
        ? DevelopmentSystem().OSName()
        #--> windows
        ? App(:phone).System().OSName()
        #--> android
        ? App(:sensor).System().OSName()
        #--> espidf
        ? @@( App(:sensor).System().System().Capabilities() )
        #--> [ "gpio", "network", "clock", "threads" ]
        ? CurrentSystem().IsRuntime()
        #--> 1
        ? App(:backend).System().Spawn("cron")
        #--> native
        try
                App(:phone).System().Spawn("cron")
        catch
                ? cCatchError
        done
        #--> scope 'android' (android) forbids 'process' -- spawn 'cron'. This target has no such
capability.
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le nom du système de développement est celui qui a été déclaré, donc il ne dépend pas de la machine. La ligne `native` dépend de l'hôte : elle ne tient que sur un hôte qui peut lancer un processus, comme le chapitre 31 le dit de son propre mot. Le refus d'Android, `scope 'android' (android) forbids 'process'`, et la liste de capacités de la carte sont ceux de la cible, quel que soit l'hôte. Les trois portées sont `new stzDevSystem()` ou `DevelopmentSystem()` pour le développement, qui est de l'outillage seulement ; `App(:x).System()` pour le déploiement, où le code des fonctionnalités est écrit et vérifié ; et `new stzCurrentSystem()` ou `CurrentSystem()` pour l'exécution courante, qui est ce qui tourne maintenant. L'objet portée n'a pas de `Capabilities()` propre : l'appel passe par `.System().System()`. Une demande.

L'artefact de l'architecte est un fichier texte. Peut-il être relu, et que dit la plateforme d'une solution qui n'est pas saine ? Exécutez-le :

HARO

```
oSol = new stzPlatformProfile("greenhouse")
oSol {
        DevelopedOn(:Windows)
        WithServer(:backend, :LinuxServer)
        WithSuperApp(:phone, :Android)
        WithApp(:sensor, :ESP32)
        ? ToStzPlatform()
        #--> # .stzplatform -- a Softanza solution profile
        #    platform: greenhouse
        #    developed_on: windows
        #    server: backend -> linux
        #    superapp: phone -> android
        #    app: sensor -> espidf
}
oBack = StzPlatformProfileQ()
oBack.FromString( oSol.ToStzPlatform() )
? oBack.App(:sensor).DeploymentOSName()
#--> espidf
oBad = new stzPlatformProfile("empty")
? @@( oBad.Validate() )
#--> [ "no development system declared (use DevelopedOn)", "the solution deploys no apps" ]
oPlat = StzPlatformQ("empty")
oPlat.SetProfile(oBad)
try
    oPlat.Build()
catch
    ? cCatchError
done
#--> stzPlatform.Build: the solution is not sound -- no development system declared (use
DevelopedOn); the solution deploys no apps
```

✓ exécuté : a affiché ce que ce bloc promet

La forme fichier est un texte simple que l'architecte possède, et elle fait l'aller-retour depuis une chaîne. La solution malsaine est refusée avant la construction, ses deux défauts nommés : `the solution is not sound -- no development system declared (use DevelopedOn); the solution deploys no apps`. L'ordre de construction et de déploiement est celui du chapitre 33, et la barrière sur qui déploie est celle du chapitre 24.

Qui est l'architecte ? Le dossier nomme l'architecte deux fois : dans la System Foundation, et dans une note du domaine de projets pour une école qui certifierait des architectes de Softanza, avec trois niveaux, une note marquée retirée le jour

même où elle a été écrite. Cette certification est une proposition du 30 septembre 2026, non une livraison, et le site ne doit pas l'annoncer. Le protocole du domaine de projets définit un rôle comme un profil tenu dans une portée. Appliqué ici, ce qui est mon application et non celle du dossier, l'architecte n'est pas un rang. C'est ce que l'on tient dans la portée d'une solution : le profil, les mondes, la gouvernance. Je tiens cela pour une vue, non pour un résultat. Le site, tel que je l'ai lu le 9 octobre 2026, a une page de parcours pour l'architecte de plateforme et une page de certification qui dit que la proposition n'est pas annoncée. Je raconte l'architecte comme ce que l'on déclare, et la certification comme cette proposition ; aucune école n'est nommée ici.

Sept refus se tiennent derrière cette forme.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé l'hôte et la mini-appli. Des mondes, non des mini-applis ; un graphe gouverné, non un conteneur hôte. Le chapitre 4 l'a raconté, et je renvoie à lui.

J'ai refusé de faire du socle commun un genre particulier de nœud. Dans la conception, c'est un monde, le monde zéro, uniforme et récursif ; dans le code, comme la section 3 l'a imprimé, c'est un objet plateforme et non un nœud du graphe.

J'ai refusé le bac à sable comme définition de l'isolation. L'isolation est un sens déclaré et inspectable, et non une machine virtuelle en boîte noire. Un corps souverain de monde plus des liens barrés par une norme. Un bac à sable d'exécution n'est qu'une implémentation pour des mondes non dignes de confiance.

J'ai refusé de faire de la gouvernance une facette de plus, à côté de la constellation, des liens, du socle commun, de l'atteinte et de la présence de la conception. Elle est l'ambiant. Dans le code, une gouvernance par constellation, configurée par elle, de sorte qu'il y ait une gouvernance vivante et jamais une copie périmée tenue par un appelant.

J'ai refusé de donner l'enveloppe au monde. `stzApp` modélise le monde, et manque délibérément de l'enveloppe opérationnelle.

J'ai refusé de figer *courant* sous un meilleur nom. Le remède n'est pas un meilleur nom : chaque acte système est écrit dans une portée nommée. Les tentatives qu'elle refuse sont une paire de mots, ce système et le système cible, et un triplet, hôte, runtime et cible.

J'ai refusé de déployer un profil. Nous ne déployons pas un profil ; nous déployons une plateforme et ses parties. Le profil ne fait que décrire. La plateforme le possède et conduit la construction puis le déploiement.

Le tribunal dit maintenant ce qui, de tout cela, tourne.

5. LE TRIBUNAL

`06_stzsuperapp_narrated` prouve que les mondes s'enregistrent comme nœuds ; que le socle commun tient une identité et un magasin ; qu'une traversée est refusée pour un risque non déclaré, acceptée quand elle est gouvernée, et refusée quand elle n'est pas liée, avec le lien nommé ; qu'un monde retiré rompt ses liens et revit ; qu'un échange garde le nœud et le lien ; et qu'une constellation s'emboîte dans une autre. `01_platform_narrated` prouve que des coques sont générées selon l'atteinte déclarée et qu'un monde sans atteinte est refusé ; qu'une capacité n'est accordée que lorsque la gouvernance couvre son niveau de risque, et qu'une capacité jamais demandée est refusée avec une raison. Elle prouve que le socle commun enregistre des identités, ouvre des sessions, poste et lit des messages, et stocke. Elle prouve que le monde est servi sur un vrai port, et que le registre refuse un appel vers un monde non lié ou retiré. Elle écrit des coques à côté d'elle et ouvre un port de boucle locale. `full_scope_model_narrated`, que la garde appelle le socle commun de l'architecte, prouve que la solution est modélisée d'abord, qu'une solution malsaine est signalée, que les trois portées sont distinctes, que la portée refuse ce que la cible interdit et répète ce qui manque à l'hôte, que le fichier texte fait l'aller-retour, que la construction précède le déploiement, et qu'un déploiement par un acteur modèle de langage est refusé. `04_stzapp_pursue_narrated` et `05_stzapp_live_narrated` portent le but et les réactions.

La garde que le chapitre n'a pas : aucune ne juge les vérités d'un monde, un flux, ou le moment d'une réaction.

Je place maintenant le dossier à côté du terrain.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`06_stzsuperapp_narrated` 21 assertions
base/test/app/06_stzsuperapp_narrated.ring

`01_platform_narrated` 31 assertions
base/test/platform/01_platform_narrated.ring

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

`04_stzapp_pursue_narrated` 11 assertions
base/test/app/04_stzapp_pursue_narrated.ring

`05_stzapp_live_narrated` 11 assertions
base/test/app/05_stzapp_live_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

La super-appli mène en utilisateurs. La conception nomme son modèle et le remplace ; le chapitre 4 dit que le dossier ne mesure rien contre eux. La page Plateformes imprime l'autre moitié : les plateformes de clients tournent aujourd'hui sur des piles web ordinaires. Une plateforme de client sur le moteur, comme constellation, est la direction ; le chapitre 33 la porte comme la ligne suivante.

Le terrain mène en coques. La conception promet une application web progressive, une coque de bureau, une coque mobile. Le code génère une page et un lanceur de quelques centaines d'octets. Les coques sont conçues. La génération est construite, et minuscule.

Le terrain mène en ce que porte un contexte borné : une équipe et un langage. Un monde porte un graphe et une narration. Le persona pour lequel est fait un monde n'est pas un champ de la classe : un monde a la liste des surfaces qu'il atteint et rien d'autre sur qui il sert. Les cinq choses du site, un sol partagé, un monde pour chaque personne et appareil, des liens, des règles, des rôles, sont vraies du socle commun, des liens et de la gouvernance. Les rôles sont les acteurs, les permissions et l'autorité de la gouvernance. Il n'y a pas de classe rôle, et un monde par personne et appareil est une convention de nommage.

L'Atlas ne note pas ce volet. `stzApp` y est nommé une fois, dans le couloir du domaine des interfaces intitulé coque d'application et fenêtrage. C'est ce couloir, non le monde, qui est noté solide, avec la note qu'il n'y a ni orchestration multi-fenêtres, ni dialogues, ni menus, ni zone de notification. La plateforme dans un couloir du domaine du système. La constellation dans aucun.

Le compilateur mène sur les portées. Il vérifie contre une cible en usage quotidien à grande échelle ; la portée de Softanza est un objet d'exécution qu'un programme interroge. Ce qu'elle ajoute est un verdict au site d'appel dans les mots propres de la cible, et un jumeau qui répète ce qui manque à l'hôte, ce qui est le chapitre 24.

Là où le terrain et le dossier se rencontrent : une super-appli est un graphe de graphes. La carte de contextes du terrain est un diagramme. Celle de Softanza est un registre qui refuse.

7. LA LIMITE

CONSTRUIT — un monde comme déclaration avec un être, un devenir et une manière d'être rencontré ; un but dont l'écart est mesuré ; une constellation de mondes avec des liens et trois barrières sur chaque traversée ; le socle commun comme plateforme ; la jointure des capacités ; le profil d'une solution comme fichier texte, et les trois portées du système

CONÇU — l'idiome quasi naturel de la constellation ; un appel acheminé dans une constellation emboîtée ; des coques qui embarquent le moteur ; la certification de l'architecte, qui est une proposition ; la fiche de garanties de cinq lignes, comme exigence

ABSENT — un monde qui juge ses propres vérités ; une narration pour la constellation ou pour l'enveloppe de la plateforme ; une fiche de garanties de cinq lignes écrite pour une solution

Les vérités d'un monde sont narrées et non jugées. Le nombre dans une réaction en est le premier cas. Quatre-vingt-dix jours et un jour donnent-ils des propositions différentes ? Une réaction se déclenche pour chaque instance qui n'a pas de relation nommée d'après la chose qu'elle propose, et le commentaire au-dessus de `Pulse()` le dit. Faites tourner deux cliniques :

HARO

```
oA = StzAppQ("clinic-a")
oA {
        AddThingQ(:Patient) { Has([ :name ]) }
        AddThing(:Checkup)
        AddReactionQ(:Patient).Unseen(90, :Days) { Propose(:Checkup) }
        AddInstance("alice", :Patient)
        AddInstance("bob", :Patient)
        Relate("alice", :checkup, "checkup-1")
}
oB = StzAppQ("clinic-b")
oB {
        AddThingQ(:Patient) { Has([ :name ]) }
        AddThing(:Checkup)
        AddReactionQ(:Patient).Unseen(1, :Days) { Propose(:Checkup) }
        AddInstance("alice", :Patient)
        AddInstance("bob", :Patient)
        Relate("alice", :checkup, "checkup-1")
}
oA.Live()
#--> [clinic-a] is live -- 2 thing(s), 0 flow(s), 1 reaction(s), 0 goal(s); 1 proposal(s)
oB.Live()
#--> [clinic-b] is live -- 2 thing(s), 0 flow(s), 1 reaction(s), 0 goal(s); 1 proposal(s)
? @@( oA.Proposals() )
#--> [ [ "propose", "checkup", "for", "bob" ] ]
? @@( oB.Proposals() )
#--> [ [ "propose", "checkup", "for", "bob" ] ]
```

✓ exécuté : a affiché ce que ce bloc promet

Les deux cliniques répondent `[ "propose", "checkup", "for", "bob" ]`. Quatre-vingt-dix jours et un jour donnent les mêmes propositions !

La même lecture vaut pour `IsTrue`, qui est stocké et narré alors que les instances ne portent pas de champs contre lesquels le juger. Elle vaut pour le `Require` et le `Then` d'un flux, stockés et narrés, où je n'ai trouvé aucun exécutant ; et pour les bornes d'un raffinement. Un monde déclare aujourd'hui qu'un solde ne doit pas être négatif. Il ne juge pas un solde. Ce qu'il juge, c'est l'écart d'un but, une traversée et une capacité. Donc je dis « un monde qui veut » pour le but, et « narré » pour la vérité. Le `true when balance @ >= 0` du chapitre 4 est une déclaration seulement. Le chapitre le dit donc en une ligne : déclaré et narré ; jugé seulement pour les buts, les traversées et les capacités.

La conception écrit plus que la classe ne parle. La classe parle-t-elle l'idiome quasi naturel de la conception pour la constellation ? Demandez-lui `Hosts`, `Admits` et `Explain` :

HARO

```
oSuper = new stzSuperApp("acme-group")
try
    oSuper.Hosts(:VisitApp)
catch
    ? cCatchError
done
#--> Error (R14) : Calling Method without definition: hosts
try
    oSuper.Admits(:VisitApp)
catch
    ? cCatchError
done
#--> Error (R14) : Calling Method without definition: admits
try
    oSuper.Explain()
catch
    ? cCatchError
done
#--> Error (R14) : Calling Method without definition: explain
```

✓ exécuté : a affiché ce que ce bloc promet

Trois fois la réponse est `Calling Method without definition`. `Shares`, `Provides`, `Norm`, `Reaches` et `Generate` ne sont pas non plus définis sur une constellation. La classe a `AddWorld`, `AddConstellation`, `Bond`, `Retire`, `Revive`, `Swap`, la famille `Gov*`, `CallAcross`, `Crossings`, les appels du socle commun, et des

appels qui regardent : `Why`, `GraphQ`, `WorldQ`, `KindOf`, `IsActive`, `HasWorld`, `NumberOfWorlds`, `AreBonded`, `CrossingCount` et `LastCallMs`. Les verbes de composition sont ceux-là. Les trois aspects de la conception s'atteignent aujourd'hui par `AddWorld`, `Bond` et `OpenCommonsOn`, et la gouvernance par `Gov*` ; l'atteinte et la présence sont la partie qui n'est pas là. L'en-tête de la conception appelle elle-même le document une conception à venir.

Ce qui est construit : le monde ; la constellation avec ses barrières, son registre de traversées avec les durées, une décision écrite sur une traversée autorisée, et une traversée refusée notée comme preuve ; la jointure des capacités de la plateforme, son socle commun, son registre, ses coques, son profil avec construction puis déploiement ; les trois portées.

Ce qui est conçu : l'idiome quasi naturel ; un appel acheminé dans une constellation emboîtée (aujourd'hui une traversée est un verdict, et rien n'est remis à la cible) ; charger un monde sans reconstruire un hôte ; des coques qui embarquent le moteur ; la fiche de garanties de cinq lignes, que la vision demande et que la page de parcours du site marque conçue : une ligne pour chacun de cinq systèmes auxquels la solution répond, technique, social, écologique, économique et culturel.

Ce qui est absent : une narration pour la constellation ou pour l'enveloppe, puisqu'aucune n'existe. La narration pour construire une première appli utilise des verbes que la classe n'a pas. Sept de ses appels échouent à ce commit, et je la cite pour l'idée et non pour la syntaxe. Une plateforme de client sur le moteur comme constellation. La fiche de garanties de cinq lignes écrite : aucune solution n'en a une. Un couloir de l'Atlas pour le monde, la constellation ou l'enveloppe.

Les six portes viennent ensuite.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous déclarez un monde avec des accolades, vous lui demandez de s'expliquer, et vous traversez entre les mondes avec un appel qui renvoie 0 et un `Why()` qui nomme la barrière. Surveillez la copie : configurez, puis enregistrez.
L'analyste	Vous déclarez des choses, des relations et un but comme un état voulu du graphe, et l'écart est mesuré. Vos vérités et vos durées sont narrées et pas encore jugées.
Le concepteur	Un monde déclare des écrans comme des intentions, un raffinement comme un bouton borné, et où il apparaît. Les coques générées aujourd'hui sont des bouchons, et le persona qu'un monde sert n'est pas un champ.
Le décideur	Vous déclarez qui peut traverser où et à quel risque, en trois barrières, et chaque traversée autorisée laisse une décision : le bloc a imprimé sa justification, et son enregistrement tient l'autorité et le risque de son instant. Le fichier de votre architecte est un texte que vous possédez.
Le formateur	Le dossier est mince. Le plan d'éducation liste une échelle du praticien à l'architecte puis au contributeur parmi cinq schémas de niveaux concurrents, et je le dis. L'exercice ci-dessous se juge en l'exécutant.
L'agent	Un monde peut être enveloppé comme un agent d'IA que l'hôte fait avancer d'un tick à la fois. Un déploiement par un acteur modèle de langage est refusé, et rien n'est déployé. La traversée entre les mondes est gouvernée par le nom et l'autorité du monde appelant.

9. EXERCICE

Déclarez deux mondes pour un lieu que vous connaissez : une boutique et son fournisseur, une classe et sa bibliothèque, une clinique et sa pharmacie. Donnez à chacun une chose. Reliez-les par un lien pour un acte. Avant d'exécuter quoi que ce soit, écrivez ce que vous attendez que `CallAcross` renvoie, et quelle raison vous vous attendez à lire d'abord.

HARO

```
? YourConstellation.CallAcross("a", "b", "act")
#--> 1
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Puis demandez l'appel, lisez la raison, et réparez la déclaration, non le tribunal, une barrière à la fois, jusqu'à ce qu'il renvoie 1. Comptez les refus. Quelle barrière auriez-vous oubliée sans le tribunal ? Le tribunal vous dit quelle barrière refuse : un niveau de risque manquant, une permission manquante, une autorité qui ne couvre pas le risque. Il n'écrit pas la déclaration. Un exercice plus dur : retirez le second monde et prédisez la nouvelle raison avant de la demander ; puis enregistrez une copie du premier, changez votre original, et prédisez ce que voit la constellation.

Je vous laisse celui-ci. Prenez le bloc des barrières de la section 3 et supprimez une ligne : `oCon.GovSetAuthority("resto", :Delegated)`. Arrêtez le bloc à son dernier `Why()`, comme ici, et je n'imprime rien dessous :

HARO

```
oResto = StzAppQ("resto")
oResto.AddThingQ(:dish) { Has([ :name, :price ]) }
oSupplier = StzAppQ("supplier")
oSupplier.AddThingQ(:crate) { Has([ :produce, :qty ]) }
oCon = new stzSuperApp("acme-group")
oCon.AddWorld("resto", oResto)
oCon.AddWorld("supplier", oSupplier)
oCon.Bond("resto", "supplier", "order-produce")
? oCon.CallAcross("resto", "supplier", "order-produce")
? oCon.Why()
oCon.GovDeclareRisk("order-produce", 2)
? oCon.CallAcross("resto", "supplier", "order-produce")
? oCon.Why()
oCon.GovGrant("resto", "order-produce")
? oCon.CallAcross("resto", "supplier", "order-produce")
? oCon.Why()
? oCon.CallAcross("resto", "supplier", "order-produce")
? oCon.Why()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le et lisez le dernier `Why()`. Nommez la barrière qui refuse, et dites ce que le bloc devrait dire pour que la traversée soit autorisée. La section 3 dit ce que demande chacune des trois barrières, et que chaque refus nomme sa barrière. Quand vous l'aurez résolu, vous saurez que le tribunal nomme la barrière qui refuse et n'écrit pas la déclaration.

Le chapitre suivant dit comment une solution déclarée atteint la réalité, par exactement une porte : répéter sur ce qui ne touche à rien, valider par un acteur gouverné, et laisser la production refuser tout ce qui est faux. Répéter, puis valider.

OÙ SONT LES PREUVES

NARRATION	`stzApp-building-your-first-app-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
NARRATION	`stz-built-on-real-world-need.md`
SUR LE SITE	platforms
SUR LE SITE	offering
SUR LE SITE	forged
DÉCISIONS DE L'ÉTUDE	12.2, 12.12, 12.13, 12.15, 12.19, 12.25, 12.29

PARTIE V · CHAPITRE 24

Répéter, puis valider

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 11 bloc(s) sur 11 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends les trois pratiques que le terrain tient à côté du programme, planifier puis appliquer, le double de test et l'émulateur, et je montre le seul chemin que la bibliothèque met à leur place. Je répète sur un jumeau des fichiers et un jumeau de l'environnement du processus, et je lis le plan de placement. Je construis le bundle de l'émulateur, j'exécute un déploiement qui échoue et revient en arrière, et je code contre des services par leur nom avant de m'abonner à aucun. Puis je demande à la production de refuser, et j'imprime le seul faux qu'elle ne refuse pas : un double qui ne dit rien de lui-même. Le principe vient de trois documents du dossier. On atteint la réalité par exactement une porte : répéter sur ce qui ne touche à rien, valider par un acteur gouverné, et laisser la production refuser tout ce qui est faux.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Trois pratiques du terrain se tiennent à côté du programme, chacune avec sa propre configuration.

La première est planifier puis appliquer : l'infrastructure comme code et les chaînes de déploiement. Les références propres du dossier sont le graphe de dépendances des tâches d'une chaîne, le graphe de ressources d'un outil avec appliquer-ou-annuler, et un ordonnanceur de grappe qui place le travail selon les ressources qu'il

demande. L'habitude du terrain est que le plan est un fichier pour un outil, et que savoir si le mauvais acteur peut l'appliquer relève d'une politique. Le dossier énonce la perte à laquelle il répond : c'est une convention, appliquée par l'espoir.

La deuxième est la virtualisation de service et le double de test : le mode test d'un fournisseur avec des cartes de test publiées, un serveur bouchon pour HTTP, et les bibliothèques qui enregistrent un appel une fois et le rejouent. L'habitude du terrain est que le double vit dans la suite de tests, et que la production est un autre fichier. La plainte du dossier : aucune passerelle ne refusera une carte parce que vous l'avez demandé poliment.

La troisième est l'émulateur et le simulateur : un émulateur de téléphone, un émulateur de carte, un substitut local d'un nuage. La phrase propre du dossier sur le genre est que les simulateurs mentent. Ils feignent la base de données, bouchonnent l'appareil, approximent le calcul.

Ce que les trois ont en commun, c'est que chacun est un outil séparé, et que le programme lui-même n'en sait rien. C'est la connaissance générale des outils, et non une étude du dossier, sauf là où je cite les documents.

2. LE PREMIER PRINCIPE

On atteint la réalité par exactement une porte : répéter sur ce qui ne touche à rien, valider par un acteur gouverné, et laisser la production refuser tout ce qui est faux.

Le premier principe du dossier pour ce terrain est la première clause, et l'étiquette propre de la bibliothèque l'appelle un premier principe : on atteint la réalité par exactement une porte. Le chemin par défaut d'un changement qui a des conséquences est de répéter dans un jumeau, de générer un plan, et de valider le plan. Le plan est la porte. Le résumé du cadre tient en trois mots : répéter, planifier, valider. La ligne de structure du plan de ce livre dit la dernière clause : la production refuse un faux. La première clause est celle d'un document de la bibliothèque, la System Foundation, écrit sans ligne d'auteur. La clause du milieu, valider par un acteur gouverné, est celle de la conception du déploiement : "only an effectful actor commits". Le chapitre joint les trois et attribue chaque partie à sa source.

3. LA FORME SOFTANZA

LE JUMEAU

Le seul artefact qui traverse vers la réalité est un plan. Il nomme chaque étape. Il signale celle qui retire quelque chose. Et si on lui demande de vérifier, il relit la réalité. Le chapitre 20 a dit « je l'ai vu avertir » sans montrer les mots. Que dit le plan de la source qu'il allait déplacer ? Voici les mots, et rien n'est exécuté :

HARO

```
oBench = new stzVirtualFileSystem()
oBench {
        CreateFolder("config")
        CreateFile("config/pump.ini", "threshold=40")
        MoveFile("old_settings.txt", "config/legacy.ini")
}
oPlan = oBench.GenerateUpdatePlan()
oPlan {
        ? Narration()
        #--> Update plan (3 of 3 operations to commit):
        #-->   * 1. create folder 'config'
        #-->   * 2. create file 'config/pump.ini' (12 bytes)
        #-->   * 3. move 'old_settings.txt' -> 'config/legacy.ini'
        ? @@(Risks())
        #--> [ [ 3, "MOVES (removes source): move 'old_settings.txt' -> 'config/legacy.ini'" ] ]
        ? @@(Validate())
        #--> [ [ 3, "source absent in reality: old_settings.txt" ] ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le plan répond `source absent in reality: old_settings.txt`, et il signale le déplacement : `MOVES (removes source)`. La narration de la bibliothèque imprime une ligne finale, selon laquelle la réalité reste intacte jusqu'à ce que je choisisse de valider ce plan. L'exécution ne l'imprime pas, donc je ne la montre pas.

Le jumeau n'est pas seulement un jumeau de fichiers. Une variable posée, un répertoire changé et un processus enfant mis en file sont enregistrés, narrés et signalés, et aucun processus n'a démarré. L'environnement réel a-t-il la variable ? Demandez-le-lui, après que le jumeau l'a posée :

HARO

```
oEnv = new stzVirtualEnvironment()
oEnv {
        SetActor("agent-42")
        SetVar("SHOP_MODE", "rehearsal")
        Cd("/srv/shop")
        Spawn("backup-database --all")
        ? Var("SHOP_MODE")
        #--> rehearsal
        ? NumberOfPendingSpawns()
        #--> 1
}
? (new stzEnvironment()).Has("SHOP_MODE")
#--> 0
oEnvPlan = oEnv.GenerateUpdatePlan()
? oEnvPlan.Narration()
#--> Update plan (3 of 3 operations to commit):
#-->   * 1. set env var 'SHOP_MODE' = 'rehearsal'
#-->   * 2. change directory to '/srv/shop'
#-->   * 3. spawn process: backup-database --all
? @@(oEnvPlan.Risks())
#--> [ [ 3, "RUNS an external command (side effects): spawn process: backup-database --all" ] ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le jumeau détient `rehearsal` et un processus en attente, et l'environnement réel répond `0` ! Le chapitre 20 compte les deux jumeaux comme construits, et voici le second.

Le jumeau a un second travail. Un programmeur sur un portable sans broches écrit du code de broches pour une carte. La portée dit *répété*, comme le chapitre 8 l'a montré et comme le chapitre 31 le montre. Au déploiement, le pont d'abaissement transforme la répétition en source de firmware. Que se passe-t-il quand un assistant demande à déployer ? `Build()` enregistre la commande de construction et n'exécute pas la chaîne d'outils, et `Deploy` abaisse en mémoire, donc aucun fichier n'est écrit :

HARO

```
oProduct = new stzPlatformProfile("greenhouse")
oProduct {
        DevelopedOn(:Windows)
        WithApp(:sensor, :ESP32)
        ? ReadPinIn(:sensor, 34)
        #--> rehearsed
```

```
        ? WritePinIn(:sensor, 26, 1)
        #--> rehearsed
}
oPlatform = StzPlatformQ("greenhouse")
oPlatform {
        SetProfile(oProduct)
        Build()
        try
                DeployAs(LLMActor("copilot"))
        catch
                ? cCatchError
        done
        #--> stzPlatform.Deploy: actor 'copilot' lacks the 'effectful' capability -- it may
propose a deploy, not perform it.
        ? IsDeployed()
        #--> 0
        DeployAs(HumanActor("me"))
        ? ArtifactFor(:sensor)
        #--> // firmware generated by the Softanza lowering bridge
        #--> // target: espidf
        #--> void setup() {}
        #--> void loop() {
        #-->   digitalRead(34);
        #-->   digitalWrite(26, 1);
        #--> }
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'assistant est refusé, et on lui dit qu'il `may propose a deploy, not perform it` ! Le déploiement de la personne produit le firmware. La limite de ce bloc est à la section 7 : seuls deux verbes sont abaissés.

Le jumeau répète ; ensuite, le plan dit où irait chaque partie.

LE PLAN DE PLACEMENT

Avant qu'un octet soit construit, le planificateur de livraison énonce, pour chaque capacité dont une partie a besoin, si la cible la fournit, si le moteur la fournit, si une construction la fournit, ou si le serveur la fournit. Il donne la raison en une phrase. Le chiffre entre parenthèses est l'estimation du catalogue et non une mesure, donc ne le citez pas comme une affirmation. « 1 parts » est la grammaire propre de la bibliothèque :

HARO

```
oShip = new stzDelivery("shop")
oShip {
        AddSuperApp(:phone, :Android)
        NeedsIn(:phone, [ :Unicode, :PivotTable, :Collection, :Neural ])
        Plan().Show()
        #--> Solution 'shop' -- placement & scope plan (rehearsal; nothing built yet)
        #--> ========================================================================
        #-->
        #-->   Part 'phone' [superapp] -> android (mobile / edge)
        #-->      Unicode            [target]    Unicode: the target-platform is strong at it ->
use its own
        #-->      PivotTable         [stz.wasm]  PivotTable: Softanza-differential, mobile weak -
> the on-device engine
        #-->      Collection         [stz.js]    Collection: ergonomic -> a Softanza construct in
the target language
        #-->      Neural             [server]    Neural: heavy -> runs on the backend, not the
edge
        #-->      -> stz.wasm carries: [ "PivotTable" ]  (~12 KB, 1 of 4)
        #-->
        #-->   Summary: 4 capabilities across 1 parts -> target 1, engine 1, construct 1, server
1.
        #-->   Build() will compile exactly this scope; Deploy() will commit it.
}
```

✓ exécuté : a affiché ce que ce bloc promet

La règle derrière le plan, dans la conception de l'émulation : la périphérie ne porte une capacité que si elle est critique, et soit propre à Softanza, soit faible ou absente sur la plateforme cible. Une expression régulière est une construction posée sur celle de la plateforme, et jamais le module de correspondance du moteur compilé pour le navigateur.

Le plan dit où ; l'émulateur construit ce que dit le plan.

L'ÉMULATION

L'émulateur transforme le plan en un bundle, et le bundle est ce qui est livré. Dans le bloc suivant, le bundle généré pour la répétition est câblé directement au rattachement de production, et c'est l'acteur qui décide. Que devient le site quand un assistant appelle `Deploy(:Production)`, et que devient-il quand une personne le fait ? Le manifeste qui arrive nomme le groupe du moteur que la partie porte. La

cible ici est un répertoire local, le genre de site qui n'a besoin d'aucun outil hors des opérations sur fichiers. Le bloc écrit deux répertoires dans le répertoire de travail, `_bundle` et `_live`, et les retire à sa fin :

HARO

```
cHere = StzReplace(WorkingDirectory(), "\", "/")
oFront = new stzDelivery("shopfront")
oFront.AddSuperApp(:web, :Browser)
oFront.NeedsIn(:web, [ :Unicode, :PivotTable, :Collection ])
oBundle = new stzEmulator(oFront)
oBundle {
        CompileEngineQ(FALSE)
        SetOutDirQ(cHere + "/_bundle")
        Build()
}
oSite = new stzDeploymentSite("cdn")
oSite.SetKindQ(:LocalRepo).SetStoreAtQ(cHere + "/_live")
oFront {
        DeployTo(oSite, :web)
        ShipSlice(:web, oBundle)
        SetActor(LLMActor("assistant"))
        Deploy(:Production)
        ? oSite.Status()
        #--> absent
        SetActor(HumanActor("ops"))
        Deploy(:Production)
        ? oSite.Status()
        #--> launched
}
? @@(StzEngineDirListFiles(cHere + "/_live"))
#--> [ ".stzsite", "deploy.json", "index.html", "stz.js" ]
? read(cHere + "/_live/deploy.json")
#--> {
#-->   "solution": "shopfront",
#-->   "part": "web",
#-->   "engine": "aggregation",
#-->   "artifacts": "index.html,stz.js"
#--> }
StzDirDeleteAll(cHere + "/_bundle")
StzDirDeleteAll(cHere + "/_live")
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

L'assistant a laissé le site `absent`, et la personne l'a rendu `launched`. La moitié honnête : la ligne des artefacts du manifeste dit `index.html,stz.js`, et aucun fichier du moteur pour la partie web n'y figure, parce que la compilation du moteur était désactivée. Le bloc dit ce que l'exécution a livré, et non ce qu'une exécution complète livrerait. Un `Deploy(:Production)` nu, sans acteur du tout, renvoie une répétition et écrit un avertissement, "no effectful actor -- returning a rehearsal (nothing committed)". Je l'ai exécuté aussi, dans ma recherche du 8 octobre 2026 et non dans un bloc montré ici : le site est resté absent et rien n'a été écrit.

Je n'ai pas exécuté `Deploy(:Emulated)`, parce qu'il construit le sous-ensemble de chaque partie par la chaîne d'outils du moteur, et que la construction de cette chaîne d'outils est une chose que cette machine ne doit pas exécuter. Ce que j'ai exécuté, c'est le bundle sans la compilation. Dans ma recherche du 8 octobre 2026, hors des blocs de ce chapitre, sa carte nommait un fichier du moteur qui n'y était pas. La section 7 dit ce que cela signifie pour l'affirmation que le moteur tourne dans un navigateur.

LE DÉPLOIEMENT

Pour un site local, un déploiement qui échoue défait ce qu'il avait fait ; les autres genres de site essaient seulement, au mieux. La partie web se déploie et passe sa barrière. Le stockage de la partie api échoue. Les étapes restantes sont sautées et le déploiement ne valide pas. Que devient le site web, déjà en ligne ? Le site qui échoue est un serveur sans emplacement de stockage, ce qui échoue sans lancer quoi que ce soit. Le bloc écrit un répertoire, `_site_cdn`, et le retire :

HARO

```
cHere = StzReplace(WorkingDirectory(), "\", "/")
oShop10 = new stzDelivery("shop")
oShop10.AddBackend(:api, :LinuxServer)
oShop10.AddSuperApp(:web, :Browser)
oGood = new stzDeploymentSite("cdn")
oGood.SetKindQ(:LocalRepo).SetStoreAtQ(cHere + "/_site_cdn")
oBad = new stzDeploymentSite("prod-api")
oBad.SetKindQ(:Server)
oRun = new stzDeployment(oShop10)
oRun {
        SetActor(HumanActor("ops"))
        SetTarget(:web, oGood)
```

```
        SetTarget(:api, oBad)
        RunAfter(:api, :web)
        aRun = Run()
}
? aRun[1]
#--> 0
for aLine in aRun[2]
        ? aLine[1] + " -> " + aLine[2]
next
#--> store:web -> done
#--> launch:web -> done
#--> verify:web -> done
#--> store:api -> FAILED
#--> launch:api -> skipped
#--> verify:api -> skipped
? oGood.Status()
#--> rolledback
StzDirDeleteAll(cHere + "/_site_cdn")
```

✓ exécuté : a affiché ce que ce bloc promet

Le stockage de la partie api dit `FAILED`. Le site web, dont les trois étapes disent `done`, dit `rolledback` ! Avec un assistant comme acteur, la même exécution renvoie chaque étape comme *répétée*, le site reste absent, et aucun répertoire n'est créé. Je l'ai exécuté dans ma recherche du 8 octobre 2026, hors des blocs de ce chapitre, et je ne l'imprime pas.

Le déploiement échoue sans danger ; les services viennent ensuite.

LE CODE D'ABORD, L'ABONNEMENT ENSUITE

Le code demande un service par son nom et n'apprend jamais quelle implémentation il a reçue. Le bac à sable voit le premier appel et aucun après que j'ai relié le service en réel ; le second appel va vers l'adaptateur réel. La surface est saine dès que chaque liaison réelle nomme un secret que le magasin de secrets détient. La passerelle réelle de ce bloc est une classe de remplacement écrite ici, puisque la bibliothèque ne livre aucun adaptateur réel de passerelle de cartes. Une classe de remplacement nommée `LiveGateway` qui renvoie "authorized" est un faux en tout sauf son nom. Le registre la prend pour réelle parce qu'elle ne déclare rien, et la section 7 dit pourquoi cela compte :

HARO

```
oPay = new stzPaymentsSandbox()
oReg = new stzServiceRegistry("shop")
oReg.DeclareMany([ :payments, :mail ])
oReg.Bind(:payments, oPay)
oReg.Bind(:mail, new stzMailSandbox())
? Checkout(oReg, 4200)
#--> authorized
? oPay.NumberOfAuthorizations()
#--> 1
oStoreCf = new stzSecretStore("shop")
oStoreCf.Register( (new stzApiKey("stripe-key")).FromLiteralQ("sk-test-123") )
oStoreCf.Register( (new stzApiKey("mail-key")).FromLiteralQ("mk-test-123") )
oReg.BindLive(:payments, new LiveGateway, "stripe-key")
oReg.BindLive(:mail, new LiveMailer, "mail-key")
? Checkout(oReg, 4200)
#--> authorized
? oPay.NumberOfAuthorizations()
#--> 1
? oReg.PostureOf(:payments)
#--> live
oReg.SetPhase(:production)
? oReg.IsSoundVia(oStoreCf)
#--> 1

func Checkout poReg, nCents
        aRes = poReg.Service(:payments).Authorize(nCents, "tok_visa")
        poReg.Service(:mail).Send("dana@example.com", "Receipt", "Thank you.")
        return aRes[:status]

class LiveGateway
        def init
                return
        def Authorize nAmount, cToken
                return [ :status = "authorized" ]

class LiveMailer
        def init
                return
        def Send cTo, cSubject, cBody
                return 1
```

✓ exécuté : a affiché ce que ce bloc promet

Le jour où vous vous abonnez, vous changez une liaison et pas une ligne. La clé est une chaîne de test inventée et non une vraie.

LA PRODUCTION REFUSE

Un seul rapport nomme cinq des huit façons dont une surface n'est pas apte à la production. Quatre sont des erreurs : un service déclaré jamais lié (`unbound-service`), un faux (`sandbox-in-production`), une vraie base de données qui disparaît au redémarrage (`ephemeral-in-production`), et un adaptateur réel dont la clé n'est pas dans le magasin de secrets (`live-without-secret`). La cinquième est un avertissement : un adaptateur réel qui ne nomme aucun secret (`inline-credential`). Le chapitre 31 montre le service non lié et la levée d'erreur. Voici le reste de la liste. En phase de développement, la surface n'est déjà pas saine, puisqu'un service non lié et une clé manquante sont des erreurs dans toute phase. Les deux autres erreurs n'apparaissent qu'en phase de production. Le code liste huit invariants, et le plan des services (section 4) et la narration en listent cinq :

HARO

```
oStoreSf = new stzSecretStore("shop")
oSurf = new stzServiceRegistry("shop")
oSurf {
        Bind(:payments, new stzPaymentsSandbox())
        Bind(:scratch, StzMemoryDataSourceQ())
        BindLive(:cache, new stzString("cache-adapter"), "redis-url")
        BindLive(:sms, new stzString("sms-adapter"), "")
        Declare(:reporting)
        ? IsSoundVia(oStoreSf)
        #--> 0
        SetPhase(:production)
        ReportVia(oStoreSf)
        #--> Service registry 'shop' [production] -- 4 bound: 1 sandboxed, 1 local, 2 live
        #-->   [ERROR] unbound-service @ shop/reporting -- declared but never bound -- asking for
it will raise
        #-->   [ERROR] sandbox-in-production @ shop/payments -- still bound to a SANDBOX in a
production phase -- a fake must never ship
        #-->   [ERROR] ephemeral-in-production @ shop/scratch -- a LOCAL source that vanishes on
restart (in-memory) is bound in a production phase
        #-->   [ERROR] live-without-secret @ shop/cache -- credential 'redis-url' is not in the
secret store
        #-->   [WARN] inline-credential @ shop/sms -- live adapter bound without naming a store
secret
}
```

✓ exécuté : a affiché ce que ce bloc promet

Deux questions différentes sont posées à la porte : qui est refusé, et ce qui est refusé. Pourquoi chacun est-il refusé ? Demandez-le deux fois au registre, une fois avec un assistant et une fois avec une personne, en phase de développement :

HARO

```
oStoreWw = new stzSecretStore("shop")
oOne = new stzServiceRegistry("shop")
oOne {
        Bind(:payments, new stzPaymentsSandbox())
        ? Phase()
        #--> development
        ? MayGoLive(LLMActor("assistant"), oStoreWw)
        #--> 0
        ? WhyNotLive(LLMActor("assistant"), oStoreWw)
        #--> actor 'assistant' is not effectful -- it may propose, not commit
        ? MayGoLive(HumanActor("dana"), oStoreWw)
        #--> 0
        ? WhyNotLive(HumanActor("dana"), oStoreWw)
        #--> sandbox-in-production: still bound to a SANDBOX in a production phase -- a fake must
never ship
        ? Phase()
        #--> development
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'assistant est refusé pour ce qu'il est : `actor 'assistant' is not effectful -- it may propose, not commit`. La personne est refusée pour ce qu'est la surface : `sandbox-in-production: still bound to a SANDBOX in a production phase -- a fake must never ship`. Et poser la question laisse la phase où elle était : `development` avant, `development` après, bien que la question soit posée dans le cadre de la production.

Et le registre connaît des services alors que la livraison connaît des parties. Réunis, ils répondent à une question qu'aucun des deux ne peut traiter seul. Quelle partie est sur le point de partir sur un double ? Le registre lui-même ne signale rien, puisque la phase est le développement et qu'il n'y a pas de constats. La vérification au niveau des parties répond quand même, par destination, et nomme la partie :

HARO

```
oS3 = new stzServiceRegistry("shop")
oS3 {
        DeclareMany([ :payments, :cache ])
        BindSandbox(:payments, new stzPaymentsSandbox())
        BindLocal(:cache, StzMemoryBlobStoreQ())
}
oD3 = new stzDelivery("shop")
oD3 {
        AddBackend("api", "linux")
        AddApp("web", "browser")
        UseServicesQ(oS3)
        NeedsServiceInQ("api", [ :payments, :cache ])
        NeedsServiceInQ("web", [ :reporting ])
        DeployTo(StzDeploymentSiteQ("host1"), "api")
}
? oS3.Phase()
#--> development
? len(oS3.Findings())
#--> 0
for aF in StzCheckServiceDelivery(oD3)
        ? aF[:rule] + " @ " + aF[:where] + " [" + aF[:severity] + "]"
next
#--> production-part-uses-sandbox @ api [error]
#--> production-part-uses-ephemeral @ api [error]
#--> part-uses-undeclared-service @ web [warning]
```

✓ exécuté : a affiché ce que ce bloc promet

Elle nomme `api` deux fois, avec `[error]`, et `web` une fois, avec `[warning]`, là où le registre n'avait rien signalé !

Voilà ce que l'exécution a montré ; les décisions derrière elle viennent ensuite.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Tout ce qui suit vient des documents de conception et des narrations de la bibliothèque, écrits en sessions sans ligne d'auteur. Le dossier ne contient aucune décision de ma part pour ce chapitre. Ses décisions sont celles des documents.

J'ai refusé le simulateur qui ment. L'émulateur ne feint rien : dans les mots de la conception, chaque partie exécute le même artefact de moteur compilé qui sera livré à sa cible. Cette affirmation a une mise en garde, à la section 7 : la preuve que le moteur tourne dans un navigateur n'est pas dans l'arbre épinglé.

J'ai refusé de relivrer ce que la cible fait bien. La périphérie ne porte une capacité que si elle est critique et soit unique, soit faible sur la cible.

J'ai refusé la prolifération d'outils, et la convention appliquée par l'espoir. La cible doit être à une configuration près, et la déployer doit être un seul acte gouverné.

J'ai refusé un secret écrit en dur. Un site porte une référence d'authentification, pas une clé.

J'ai refusé de toucher à la réalité par défaut. La production ne valide que si l'acteur peut valider. Un `Deploy(:Production)` nu, sans acteur, ne touche à rien.

J'ai refusé le faux complaisant. Un faux complaisant est plus dangereux que pas de faux du tout. Je tiens cela pour une vue, non pour un résultat. Le silence est la pire réponse : le mode indulgent répond 501, parce qu'un échec de correspondance ne doit jamais pouvoir passer pour un succès.

J'ai refusé « faux contre réel » comme vérité entière. C'était trop grossier. Il y a maintenant quatre postures : bac à sable, local, réel et conformité. La posture locale a été ajoutée quand l'exemple de la base de données a montré qu'un fichier SQLite est une vraie base de données dans un fichier qu'on possède.

J'ai refusé une barrière qui échoue en laissant passer. Le registre gardait autrefois son état dans des attributs. L'attacher à une livraison en prenait un instantané, si bien que la barrière aurait laissé passer une surface non saine. Une barrière qui échoue en laissant passer est pire que pas de barrière. L'état se trouve désormais dans une table indexée par identité, de sorte que chaque copie est le registre. Tout ce qui gouverne les autres doit survivre à la copie.

J'ai refusé de virtualiser la qualité d'un modèle d'IA. La qualité d'un modèle de pointe ne peut pas être virtualisée. Ce que le bac à sable vend à la place, c'est le déterminisme, un coût sur lequel on peut faire une assertion, et le travail hors ligne.

J'ai refusé de livrer les adaptateurs réels. Le plan des services dit que l'adaptateur réel de Stripe, PayPal et Adyen n'est délibérément pas livré, puisqu'un compte, une clé et un réseau dépendent de l'infrastructure. Un adaptateur du hub est construit, et le chapitre 31 le montre. Le contrat est défini, et le registre refuse de laisser le faux partir à sa place.

J'ai refusé de poser une question dans le mauvais cadre. `MayGoLive` interrogeait autrefois sur la santé dans la phase courante, si bien qu'avec un faux lié il répondait oui. Maintenant il est interrogé dans le cadre de la production quelle que soit la phase.

J'ai refusé de dire que le besoin du restaurant était l'origine de la conception. Le site dit que les conceptions de l'émulation et du déploiement prennent une solution comme exemple. C'est un exemple, pas l'origine de la conception. La solution d'exemple des documents porte le nom de produit d'un client, et je l'ai renommée dans chaque bloc. Le chapitre 28 raconte l'histoire, et je ne la répète pas ici.

Le plan des services est intitulé « écrit le 23 juillet 2026 en réponse à la question de l'utilisateur ». La question demande si l'émulation couvre les bases de données, les API métier, les modèles de pointe et les fournisseurs de nuage. Le but est qu'un programmeur puisse coder un programme entier sans abonnements ni frais et

obtenir quand même une solution fonctionnelle prête à déployer. Le chapitre rapporte le titre du plan et sa date. Le dossier ne dit pas qui a posé la question, et je ne prétends pas qu'elle était de moi.

5. LE TRIBUNAL

`service_registry_narrated` prouve la surface déclarée, qu'un service non lié lève une erreur, et que l'application demande par nom. Il prouve qu'un bac à sable en production est une violation, avec un constat par faux, et que le justificatif doit être dans le magasin de secrets. Il prouve que passer en réel est un franchissement gouverné, avec un assistant refusé et une personne refusée à cause d'un faux, et que poser la question laisse la phase. Il prouve aussi les aides d'intégration continue, la quatrième posture, et un adaptateur réel qui a besoin d'un certificat. `service_delivery_narrated` prouve la surface externe répétée dans le plan, et le contrôle préalable. Il prouve la barrière, qui lors d'un déploiement en production refuse un faux et consigne chaque raison, et s'ouvre quand la surface est réelle. Il prouve que le registre est un seul registre quel que soit le nombre de copies faites, et la question de savoir quelle partie dépend d'un faux. Ma recherche du 8 octobre 2026 a exécuté les deux : 93 sur 93 et 60 sur 60. Le second a laissé à côté de lui un fichier de zéro octet, et je l'ai supprimé.

`virtual_system_twin_narrated` prouve, en douze scènes, que le jumeau ne touche à rien de réel, que le plan narre et signale, que la validation est l'unique porte, qu'une portée refuse, et qu'un relecteur rejette une étape. Il prouve aussi que cent opérations restent en mémoire jusqu'à une seule validation. `virtual_environment_twin_narrated` prouve la même forme pour les variables, un répertoire et des lancements de processus en file. `emulator_narrated` prouve que le bundle se génère, que la carte rend chaque partie par rôle, et que chaque partie ne porte que son sous-ensemble. `deployment_narrated` prouve le site comme configuration, la validation gouvernée, le plan d'étapes, le retour arrière, la faisabilité, et le backend git ; il lance `git`, donc le livre ne l'exécute pas.

`full_scope_model_narrated`, scènes 13 à 16, prouve que construire précède déployer, qu'une solution non saine ne peut pas être construite, le pont d'abaissement, et le déploiement gouverné.

Le refus que fait la production est prouvé par les deux premières gardes, qui ont tourné. Que le moteur répété tourne dans un navigateur est la preuve suivante : aucune garde qui tourne ne le vérifie aujourd'hui.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`service_registry_narrated` 93 assertions
base/test/system/service_registry_narrated.ring

`service_delivery_narrated` 60 assertions
base/test/system/service_delivery_narrated.ring

`virtual_system_twin_narrated` 48 assertions
base/test/system/virtual_system_twin_narrated.ring

`virtual_environment_twin_narrated` 38 assertions
base/test/system/virtual_environment_twin_narrated.ring

`emulator_narrated` 39 assertions
base/test/system/emulator_narrated.ring

`deployment_narrated` 63 assertions
base/test/system/deployment_narrated.ring

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les services réels sont en tête. Le plan le dit : un double n'est pas le service réel, et il faut encore au moins une exécution réelle avant la production. Une passerelle scriptée n'est pas l'ensemble des cas limites d'un fournisseur. Les réponses enregistrées dérivent, et une suite de rejeu demande de ré-enregistrer et de tester les contrats contre l'API réelle. La bibliothèque a l'appel d'enregistrement ; une vérification de dérive pour les réponses enregistrées est conçue.

Les modèles d'IA de pointe sont en tête, et ne peuvent pas être émulés. Le modèle d'IA est jugé contre le vrai, délibérément, comme sa propre activité.

L'infrastructure réelle est en tête. Le déploiement dépend de l'infrastructure : vérification réelle contre un hôte ou un compte réel. Les commandes sont générées, et à un hôte joignable près d'être littérales.

Les outils de déploiement matures sont en tête. La narration le dit clairement : cela ne remplace pas tous les outils de déploiement matures, cela unifie les coutures que ces outils laissent exposées.

La fidélité a une frontière déclarée. L'émulation est fidèle mais pas omnisciente, donc elle dit ce qui est quoi. Ce qu'un double prouve est plus étroit encore : un puits de capture prouve ce que vous auriez envoyé, et un répertoire n'est pas un CDN.

Là où je pense que le dossier se distingue, c'est dans le couplage. Le double, le plan, les noms de justificatifs, la barrière et l'acteur sont déclarés dans le même programme que celui qui est livré, et une seule barrière les juge.

Voilà le terrain ; la limite est l'endroit où je m'arrête.

7. LA LIMITE

CONSTRUIT

un jumeau des fichiers et un jumeau de l'environnement du processus, chacun avec son historique, ses instantanés, un plan narré, ses risques et, pour le jumeau des fichiers, une vérification contre la réalité ; le plan de placement ; le bundle de l'émulateur, câblé au rattachement de production ; un déploiement comme graphe d'étapes avec un retour arrière ; le registre des services avec ses postures et la barrière qui refuse un faux en production

CONÇU	le moteur vu tournant dans un navigateur ; une vérification de dérive pour les réponses enregistrées ; des branches d'un jumeau ; des jumeaux pour les bases de données, les interfaces et les réseaux ; la phase émulée posée par le déploiement émulé
ABSENT	tout adaptateur réel de passerelle de cartes pour les paiements (un adaptateur du hub est construit et prouvé seulement contre un jumeau, chapitre 31), un modèle d'IA, le courrier, les SMS ou un magasin d'objets ; la vérification réelle d'un hôte distant ; le regard d'une personne sur la surface de l'émulateur ; une solution client émulée et déployée par ce chemin

Un faux écrit à la main qui ne dit rien de lui-même passe la barrière. Le registre lit ce que l'objet dit qu'il est. Un double qui ne déclare ni qu'il est un bac à sable ni qu'il est local et réel est pris pour réel : pas d'avis veut dire réel. Donc « la production refuse un faux » est exact pour les doubles propres de la bibliothèque, et pour tout double qui se déclare. Quelle posture le registre donne-t-il à un double qui ne déclare rien, et la surface est-elle saine ? Demandez-le-lui en phase de production :

HARO

```
oUndeclared = new stzServiceRegistry("shop")
oUndeclared {
        Bind(:payments, new AlwaysYes)
        SetPhase(:production)
        ? PostureOf(:payments)
        #--> live
        ? IsSound()
        #--> 1
}

class AlwaysYes
        def init
                return
        def Authorize nAmount, cToken
                return [ :status = "authorized" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `live`, et `IsSound()` répond `1` : le double toujours-oui est passé ! La classe de remplacement du bloc antérieur, `LiveGateway`, est un tel objet.

L'abaissement connaît deux verbes. Seuls `read_pin` et `write_pin` deviennent du code. Toute autre opération répétée devient une ligne de commentaire. Dans ma recherche du 8 octobre 2026, hors des blocs de ce chapitre, j'ai répété `UseIn(:sensor, :gpio, "blink the status led")` et le firmware a reçu un commentaire. Pour un serveur ou un téléphone, l'artefact est un en-tête sans ligne quand rien n'a été répété. Le « la carte est le dernier pouce » de la narration vaut pour les lectures et écritures de broches seulement.

Le contrat de fidélité est par classe de cible. La conception dit que chaque partie déclare ce qui est fidèle et ce qui est approximé. Le code renvoie un texte fixe pour un microcontrôleur, qui liste deux choses approximées, et un pour tout le reste, "faithful".

Les tailles sont des estimations. Le plan imprime un chiffre tiré d'un catalogue, et la narration imprime deux chiffres différents pour la même partie. N'imprimez pas une taille comme une affirmation.

`Deploy(:Emulated)` ne déplace pas encore le registre. Le plan dit qu'il résout les services vers des bacs à sable. Le code construit l'émulateur et retourne ; seul le déploiement de production touche à la phase du registre. La valeur de phase `:emulated` que possède le registre n'est posée aujourd'hui que par une garde ; la poser depuis le déploiement émulé est conçu.

Un `Deploy(:Production)` refusé laisse le registre en phase de production. Après le déploiement refusé de ma recherche du 8 octobre 2026, hors des blocs de ce chapitre, `Phase()` a imprimé `production`. Le contrôle préalable restaure la phase, et le déploiement non.

Un plan signale aujourd'hui ; classer est la direction. Le chapitre 20 l'a dit, et le document du cadre dit "ranked risk assessment".

Ce qui est conçu : le moteur tournant dans un navigateur. Les documents disent que le `solve 2 -8` du téléphone renvoie `x = 4` dans une page. La garde qui le vérifie, `wasm_narrated`, dit dans son propre en-tête qu'elle vérifie le câblage et la

construction, puisque l'ancien langage hôte ne peut pas exécuter wasm. Elle cite une page dans le navigateur avec "8/8 checks". Cette page n'est pas dans l'arbre épinglé : je l'ai cherchée et n'en ai trouvé aucune. Et je n'ai pas vu la surface de l'émulateur dans un navigateur, et le dossier ne nomme personne qui l'ait vue. Par la loi du chapitre 33, cette surface est non perçue. Le chapitre dit donc : câblé, sa construction vérifiée par une garde que je n'ai pas exécutée, non vu dans un navigateur par quiconque le dossier nomme.

Est aussi conçu : des branches d'un jumeau et les alternatives et le retour du plan, pour lesquels aucune méthode n'est encore construite ; des jumeaux pour les bases de données, les interfaces et les réseaux ; et une vérification de dérive pour les réponses enregistrées.

Ce qui est absent : un adaptateur réel de passerelle de cartes pour les paiements (un adaptateur du hub est construit et prouvé seulement contre un jumeau, chapitre 31), un modèle de pointe, un magasin d'objets, les SMS ou le courrier. Les contrats et les doubles sont livrés, et les clients réels ne le sont pas, délibérément. La vérification réelle d'un hôte ou d'un compte distant : les commandes sont générées, et rien n'a tourné contre un hôte. Une solution client émulée et déployée par ce chemin : rien dans le dossier pour l'instant, ce que le chapitre 33 porte comme la ligne suivante.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous codez toute l'intégration contre des noms, avec des doubles qui lèvent une erreur quand vous avez oublié quelque chose et un compteur qui vous permet de faire une assertion sur un frais. Le jour où vous vous abonnez, vous changez une liaison et pas une ligne. La perte : un double que vous avez écrit vous-même et qui ne déclare rien est pris pour réel par la barrière.
L'analyste	Avant que quoi que ce soit bouge, vous pouvez lire le plan de placement, les dépendances externes avec les justificatifs que la production demandera par leur nom, le graphe d'étapes, et quelle partie dépend d'un faux. Les tailles sont des estimations, et les constats sont par partie.
Le concepteur	L'émulateur génère une surface à partir du plan, avec une grille de parties par rôle et une fenêtre par partie. J'ai vérifié le texte généré seulement, et jamais l'apparence. Dites « généré », et non « vu ».
Le décideur	« Qui peut déployer ? » est une propriété. Un assistant peut planifier la livraison et le site reste absent. Une personne peut valider. Aucun des deux ne peut livrer un faux déclaré. La perte à imprimer : les hôtes réels et les fournisseurs réels dépendent de l'infrastructure, et le dossier ne nomme aucune exécution réelle.
Le formateur	Une personne qui apprend peut construire l'intégration d'une boutique entière sans compte, sans clé et sans frais, et regarder le double refuser ce qu'elle a oublié. Son déploiement échoue sans danger et revient en arrière.
L'agent	Vous pouvez répéter toute l'intégration et planifier toute la livraison, dans un monde où rien de réel ne peut bouger : un paiement en bac à sable ne déplace aucun argent et un puits de courrier n'envoie aucun courrier. Vous pouvez demander si vous pouvez passer en réel et qu'on vous dise pourquoi non. Vous ne pouvez pas valider. La perte : cette garantie couvre les doubles propres de la bibliothèque, et une surface liée à quelque chose de réel est hors de sa portée.

9. EXERCICE

Écrivez, sous chaque ligne, ce que vous attendez avant de l'exécuter :

```
HARO
```

```
oMine = new stzServiceRegistry("my-project")
oMine.Bind(:payments, new stzPaymentsSandbox())
oMine.Bind(:notes, StzMemoryDataSourceQ())
oMine.BindLive(:maps, new stzString("maps-adapter"), "maps_key")
oStoreMine = new stzSecretStore("my-project")
# under each line, write the line you expect before you run it
? oMine.IsSound()
? oMine.IsSoundVia(oStoreMine)
? len(oMine.FindingsForProductionVia(oStoreMine))
? oMine.MayGoLive(HumanActor("me"), oStoreMine)
? oMine.Phase()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Ensuite, enregistrez une clé nommée `maps_key` dans le magasin de secrets, et écrivez de nouveau le nombre de constats de production. Puis liez une classe écrite à la main avec une méthode `Authorize` qui dit toujours oui, et demandez quelle posture le registre lui donne. Si une promesse casse, ne la changez pas : expliquez d'abord la sortie. Je ne vous dis pas où vous serez surpris.

Je vous laisse celui-ci. Le bloc ci-dessous est celui de la section 3, « Le code d'abord, l'abonnement ensuite », avec un nom changé. Je l'ai changé exprès :

HARO

```
oPay = new stzPaymentsSandbox()
oReg = new stzServiceRegistry("shop")
oReg.DeclareMany([ :payments, :mail ])
oReg.Bind(:payments, oPay)
oReg.Bind(:mail, new stzMailSandbox())
? Checkout(oReg, 4200)
? oPay.NumberOfAuthorizations()
oStoreCf = new stzSecretStore("shop")
oStoreCf.Register( (new stzApiKey("stripe-key")).FromLiteralQ("sk-test-123") )
oStoreCf.Register( (new stzApiKey("mail-key")).FromLiteralQ("mk-test-123") )
oReg.BindLive(:payments, new LiveGateway, "stripe-live")
oReg.BindLive(:mail, new LiveMailer, "mail-key")
? Checkout(oReg, 4200)
? oPay.NumberOfAuthorizations()
? oReg.PostureOf(:payments)
oReg.SetPhase(:production)
? oReg.IsSoundVia(oStoreCf)

func Checkout poReg, nCents
        aRes = poReg.Service(:payments).Authorize(nCents, "tok_visa")
        poReg.Service(:mail).Send("dana@example.com", "Receipt", "Thank you.")
        return aRes[:status]
```

```
class LiveGateway
        def init
                return
        def Authorize nAmount, cToken
                return [ :status = "authorized" ]

class LiveMailer
        def init
                return
        def Send cTo, cSubject, cBody
                return 1
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le. Comparez la réponse de sa dernière ligne à celle que la section 3 a donnée, et expliquez toute différence avant de toucher au nom ; la section 3, « La production refuse », montre comment demander son rapport au registre. Quand vous l'aurez résolu, vous saurez ce qui rend saine une liaison réelle : un secret que le magasin de secrets détient, et pas seulement un nom.

Le chapitre suivant passe des services derrière un programme à ses utilisateurs : vos utilisateurs gardent leur médium, et un nouveau médium ajoute des rendus, jamais des verbes.

OÙ SONT LES PREUVES

NARRATION	stz-service-virtualization-code-first-subscribe-later.md
NARRATION	stz-emulating-the-whole-solution-narration.md
NARRATION	stz-deploying-to-target-sites-narration.md
NARRATION	stz-planning-and-provisioning-a-deployment-narration.md
NARRATION	stz-system-dev-to-deploy-narration.md
SUR LE SITE	forged
DÉCISIONS DE L'ÉTUDE	12.13, 12.19, 12.21, 12.22, 12.25, 12.28, 12.29

PARTIE V · CHAPITRE 25

Vos utilisateurs gardent leur médium

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 11 bloc(s) sur 11 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends une phrase de la constitution de l'interface, un nouveau médium ajoute des rendus, jamais des verbes, et je l'exécute sur le médium que je peux mesurer. Une question devient une ligne de texte, part sur un bac à sable SMS, et est facturée par segment dans deux alphabets. Je montre ce que la bibliothèque livre, la question comme donnée et le tribunal, et ce que vous écrivez, le rendu et le lecteur. Je rapporte les refus du dossier comme ceux du dossier : une conversation qui n'a pas besoin de modèle d'IA, et une voix qui entend des commandes et non de la dictée. Et j'imprime la limite entière : pas d'application de messagerie, pas d'USSD, pas de ligne téléphonique, et une réponse contenant un espace, que la conversation ne peut pas encore prendre.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le commerce conversationnel, ou omnicanal : une entreprise rejoint ses clients dans les canaux qu'ils utilisent déjà, et un moteur de dialogue se tient derrière. Ce sont les pratiques, tirées de la connaissance générale et non d'une étude du dossier.

Les cadres de dialogue possèdent le modèle de dialogue. Le concepteur écrit des intentions et des flux dans la notation du cadre, et le cadre comprend le texte libre avec des modèles d'IA entraînés et livre des connecteurs vers les canaux.

La plateforme d'entreprise d'une application de messagerie : les messages arrivent par un webhook, et l'entreprise peut répondre librement à l'intérieur d'une fenêtre de service que le client a ouverte. En dehors, seulement des gabarits approuvés, par un compte d'entreprise vérifié.

Les passerelles des opérateurs : SMS et USSD. Le SMS est facturé par segment, et je le mesure moi-même plus bas. L'USSD est une session de courts écrans avec un délai d'expiration, par la passerelle d'un opérateur, pays par pays, sur n'importe quel téléphone sans données. Aucun fichier de l'instantané ne dit quoi que ce soit de l'USSD.

Et la parole de plateforme, dans le plan propre de la bibliothèque : Windows a une interface de parole, macOS en a une autre, et chaque navigateur moderne en a une troisième.

Ce qu'ils ont en commun, c'est que le dialogue est écrit une fois par canal ou une fois par cadre, et que le canal est un connecteur, un détail de déploiement. Le chapitre retourne cette image. Le monde est déclaré une fois, et le médium en est un rendu.

2. LE PREMIER PRINCIPE

Un nouveau médium ajoute des rendus, jamais des verbes.

C'est une phrase de la constitution de l'interface, imprimée sur le site : « La grammaire est fermée : un nouveau médium ajoute des rendus, jamais des verbes. » Le dossier ne dit rien sur la question de savoir si la formulation est de moi. Je la cite comme celle de la constitution. Je ne la raconte pas comme ma phrase. Les vingt-deux verbes dont elle parle sont ceux de la constitution, en six familles. L'annexe C les imprime tels que je les ai lus dans le fichier propre de la constitution le 9 octobre 2026. Le dépôt n'est pas nommé, puisqu'il n'est pas encore public.

La même phrase, construite, pour une famille de valeurs et non pour les vingt-deux verbes : la valeur est partagée, et la façon dont elle est rendue est propre à chaque médium. C'est une décision de Central, l'entité de coordination du domaine de projets, sur ma délégation, du 30 septembre 2026. Il prouve que le principe n'est pas seulement un slogan. C'est la forme de trois faces qui fonctionnent : couleur, earcon, phrase.

Pour le plancher du dialogue, la phrase propre du code : plancher déterministe, aucun modèle nécessaire. La loi du chapitre 17, appliquée à un médium.

3. LA FORME SOFTANZA

La bibliothèque livre la question comme donnée : une force, un sujet, une relation, des options et une raison. Elle livre les verbes, demander, répondre, refuser et conclure, et le tribunal. Un médium est un rendu de la donnée et un lecteur de la réponse, tous deux écrits par l'artisan du logiciel en quelques lignes. Rien dans la bibliothèque ne rend une question vers un téléphone. Ce qu'elle livre est une ligne en anglais, `AskIn`, qui est déjà un menu textuel : "(1) cheese (2) olives".

Deux petites fonctions à moi font tout le médium : l'une qui rend la question en une ligne de texte, et l'autre qui lit la réponse. La lecture compte, et je montre pourquoi dans le troisième bloc. Que coûte une question, en lettres latines et en arabe ? Exécutez-le :

HARO

```
aEn = QuestionAfter("couscous", "brik")
aAr = QuestionAfter("بريك" ,"كسكس")
? ToText(aEn)
#--> Which dish for order3? (every order needs 'dish') 1) couscous 2) brik. Reply with the
number.
? StzSmsSegments(ToText(aEn))[:segments]
#--> 1
? StzSmsSegments(ToText(aAr))[:encoding]
#--> ucs2
? StzSmsSegments(ToText(aAr))[:segments]
#--> 2
oBill = new stzSmsSandbox()
oBill {
        Send("+22790112233", ToText(aEn))
        Send("+22790112233", ToText(aAr))
        ? TotalSegments()
        #--> 3
}

func ToText(aQ)
        cOut = "Which " + aQ[:relation] + " for " + aQ[:subject] + "? (" + aQ[:why] + ")"
        for i = 1 to len(aQ[:options])
                cOut += " " + i + ") " + aQ[:options][i]
        next
        return cOut + ". Reply with the number."
```

```
func ReadReply(cText)
        for c in cText
                if NOT isdigit(c)
                        return cText
                ok
        next
        return 0 + cText

func QuestionAfter(cA, cB)
        oW = new stzKnowledgeGraph("stall")
        oW {
                Know("order1", "order")
                Know("order2", "order")
                Know("order3", "order")
                AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("order", "dish"))
                AskIn("c")
                ReplyIn("c", cA)
                AskIn("c")
                ReplyIn("c", cB)
        }
        return oW.AskInXT("c")
```

✓ exécuté : a affiché ce que ce bloc promet

La question est une donnée, donc n'importe quel médium la rend. Et la loi propre du médium, ici que le SMS est facturé par segment et que l'alphabet décide, est un test et non une surprise. Le texte anglais fait 93 caractères selon mon décompte, et le bloc imprime un segment. Le même cadre avec deux noms de plats en arabe fait 89 unités selon mon décompte (les deux noms sont plus courts de quatre lettres). Le bloc imprime `ucs2` et deux segments, parce que l'encodage change. Quatre lettres de moins, et un segment de plus ! Le numéro de téléphone est inventé, et seul l'indicatif du pays, qui est celui du Niger, est réel.

Maintenant toute la conversation, répétée sur le bac à sable SMS avant qu'aucun compte existe. Une conversation est une boucle des verbes de la bibliothèque : le médium porte chaque question vers l'extérieur et chaque réponse vers l'intérieur. Les assertions sont la facture et le refus. Que compte le bac à sable, et que dit le tribunal à "7" ? Rien dans la boucle n'est un modèle d'IA :

HARO

```
oKB = new stzKnowledgeGraph("stall")
oKB.Know("order1", "order")
oKB.Know("order2", "order")
```

```
oKB.Know("order3", "order")
oKB.AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("order", "dish"))
oSms = new stzSmsSandbox()
for cIn in [ "couscous", "brik", "7", "2" ]
        oSms.Send("+22790112233", ToText(oKB.AskInXT("c")))
        oKB.ReplyIn("c", ReadReply(cIn))
next
? oKB.ConversationQ("c").GoalState()
#--> fulfilled
? oSms.NumberOfMessages()
#--> 4
? oSms.TotalSegments()
#--> 4
? oKB.ConversationQ("c").Checkpoints()[1][:why]
#--> no option (7) was offered -- 2 option(s) on the table
? @@(oKB.Query([ "order3", "dish", "?o" ]))
#--> [ "brik" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Quatre messages, quatre segments, et la raison du tribunal mot pour mot : `no option (7) was offered -- 2 option(s) on the table`. Le refus de "7" est celui de la bibliothèque, comme au chapitre 18 ; la différence est qu'il est arrivé d'un téléphone.

Le lecteur du médium est le travail du médium. Un SMS entrant est une chaîne. La bibliothèque sépare les nombres, qui sont des indices d'options, des chaînes, qui sont des valeurs, et elle ne devine pas. Son propre commentaire : un indice ne peut jamais être pris pour une valeur littérale. Donc le lecteur du médium doit décider lequel c'est. Sans option sur la table, que devient le texte "2" ? Regardez :

HARO

```
oKT = new stzKnowledgeGraph("trap")
oKT.Know("order1", "order")
oKT.AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("order", "dish"))
oKT.AskIn("c")
? @@(oKT.ReplyIn("c", "2")[:admitted])
#--> [ "2" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Un plat nommé 2 ! Dans le bloc d'avant, mon `ReadReply` a transformé le texte "2" en nombre 2, et l'option a été choisie. C'est la phrase la plus tranchante sur le pont : un médium ajoute des rendus, et il ajoute aussi une lecture. La lecture est l'endroit où vit la culture d'un utilisateur, les chiffres, les mots, un "OUI" local. Je tiens cela pour une vue, non pour un résultat.

Une panne n'est pas un rejet. Le médium refuse dans ses propres termes. Un numéro mal formé, sans +, n'est pas un numéro E.164 : corrigez la donnée. Une panne est autre chose : réessayez. Que répond le bac à sable à chacun ?

HARO

```
oPh = new stzSmsSandbox()
oPh {
        ? Send("0612345678", "ready")[:status]
        #--> rejected
        FailNextQ()
        ? Send("+22790112233", "ready")[:status]
        #--> refused
        ? NumberOfMessages()
        #--> 0
        ? Send("+22790112233", "ready")[:status]
        #--> sent
        ? NumberOfMessages()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

`rejected` pour le numéro, `refused` pour la panne, et rien de compté. La reprise est `sent`, et le compte est de un.

Un message entrant n'est cru qu'après vérification de sa signature. Le chemin d'entrée est le motif que le volet des paiements a construit, et le vérificateur est générique, sur le corps. Aucun format d'en-tête ou de corps d'un fournisseur n'est modélisé, et le JSON est inventé. Les heures sont passées en paramètre, donc aucune horloge n'est lue. Que nomme le vérificateur quand un caractère du corps a changé depuis la signature ?

HARO

```
oSig = new stzRequestSigner("inbound")
oSig {
        AddKey("provider", "a-shared-secret")
        cBody = '{"from":"+22790112233","text":"2"}'
        cSig = SignBody("provider", cBody)
        ? VerifyWebhook(cBody, cSig, 1000, 5000)
        #--> provider
        ? "[" + VerifyWebhook('{"from":"+22790112233","text":"1"}', cSig, 2000, 5000) + "]"
        #--> []
        ? Why()
        #--> signature mismatch (tampered body, or a secret this receiver does not hold)
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le premier corps a donné `provider`. Le corps modifié a donné `[]`, et la raison est `signature mismatch (tampered body, or a secret this receiver does not hold)`. Un caractère a changé, et aucun fournisseur n'a été nommé ! Un corps rejoué à l'intérieur de la fenêtre a donné "replay detected" dans ma sonde.

Une valeur, plusieurs rendus. La forme du pont, construite pour les cinq valeurs sémantiques de la constitution de l'interface. La couleur, l'earcon et la phrase rendent une valeur. `muted` est le silence dans le son et un traitement dans la couleur. Une valeur que la loi ne connaît pas est refusée :

HARO

```
? @@(StzSemanticValues())
#--> [ "danger", "warning", "info", "success", "muted" ]
oE = new stzEarcons()
? oE.WouldFireAt("Catastrophe", 1)
#--> 0
? StzColorToNumber("danger.muted") != StzColorToNumber("success.muted")
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

La voix : dite, entendue, signifiée. Texte vers son, son vers texte, texte vers sens. Le médium appelé voix rend et lit avec une grammaire fermée, et le sens est ma déclaration. Ce bloc a besoin d'une voix française et d'un module de reconnaissance français. Cette machine parle deux langues, l'anglais et le français, et en entend une,

le français, et une machine sans elles donnerait une autre réponse, donc la sortie montrée est celle enregistrée sur cette machine. Pas de micro, pas de fichier, pas de réseau : le module de reconnaissance est celui qui tourne dans le processus. Que fait-il d'une phrase qu'on lui a apprise, et d'une phrase qu'on ne lui a pas apprise ?

HARO

```
oV = StzVoiceQ()
oV.UseLanguage("fr-FR")
oL = new stzListener()
oL {
        AcceptMeanings([ [ "disque plein", :Danger ], [ "sauvegarde terminee", :Success ] ])
        HearSound( oV.ToSoundOf("disque plein") )
        ? MeaningHeard()
        #--> danger
        HearSound( oV.ToSoundOf("je voudrais un cafe au lait") )
        ? WasNoMatch()
        #--> 1
        ? AcceptMeanings([ [ "tout brule", :Catastrophe ] ])
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

La phrase apprise a donné `danger`, la phrase non apprise n'a donné aucune correspondance (`1`), et `Catastrophe` a été refusée (`0`). La raison du refus dit que `catastrophe` n'est pas l'une des cinq valeurs sémantiques. Je n'imprime pas la confiance du module de reconnaissance, qui varie selon la machine.

Le monde déclare ses verbes et sa portée, et son code ne nomme aucun médium. Un monde contient des écrans déclarés par intention, cinq des vingt-deux verbes, et une liste de ce qu'il atteint. Rien ne refuse `:whatsapp` quand il est déclaré :

HARO

```
oApp = new stzApp("stall")
oApp {
        AddThingQ(:Dish) { Has([ :name, :price ]) }
        AddScreenQ(:Menu) { ToDiscover(:Dish)  Shows([ :name, :price ]) }
        AddScreenQ(:Basket) { ToSelect(:Dish)  Acts(:Confirm, :Order) }
        AddReaches([ :web, :whatsapp ])
        Explain()
        #--> WORLD stall   lives in: memory (not persisted)
        #      BEING
```

```
        #       dish (name, price)
        #     MET FROM WITHOUT
        #       screen menu: discover dish shows name, price
        #       screen basket: select dish
        #       reaches web, whatsapp
}
```

✓ exécuté : a affiché ce que ce bloc promet

`Acts(...)` est stocké et non narré : la ligne du panier dit `select dish`, et rien de `:Confirm`. Et la scène en exécution : que dit la plateforme quand on lui demande de générer la surface que le monde a déclarée ? Ce bloc crée un répertoire vide dans le répertoire de travail :

HARO

```
oPlat = StzPlatformQ("stall-platform")
oPlat.SetWorld(oApp)
try
        oPlat.Generate(:whatsapp)
catch
        ? cCatchError
done
#--> stzPlatform.Generate: unknown reach surface 'whatsapp' (web/desktop/mobile).
```

✓ exécuté : a affiché ce que ce bloc promet

`unknown reach surface 'whatsapp' (web/desktop/mobile)` : la plateforme l'a refusée ! Elle nomme les trois surfaces qu'elle a, et le même refus répond à `:ussd`.

Ce qu'on peut dire à un client est un ensemble fermé. Le tribunal de ce qui est dit a aujourd'hui un mécanisme : une énumération fermée compilée en grammaire. Les chapitres 11 et 19 l'utilisent pour un langage et pour la sortie d'un modèle d'IA. Ici elle contient les réponses qu'une entreprise accepte d'envoyer. Ce n'est pas une unité de gabarit de messagerie, et il n'y en a aucune. Que dit le tribunal d'une phrase que l'entreprise n'a jamais approuvée ? Deux choses à imprimer avec elle : le message dit "the model answered" quel que soit le locuteur, et l'énumération est insensible à la casse :

HARO

```
oSay = StzOutputSchemaQ([ [ :field = "reply", :type = :oneof, :choices = [ "Your order is
received.", "Your order is ready.", "We are closed tonight." ] ] ])
oSay {
        ? ParseOutput('{"reply": "Your order is ready."}')[:ok]
        #--> 1
        aNo = ParseOutput('{"reply": "Your order is free, tonight only."}')
        ? aNo[:ok]
        #--> 0
        ? CiteFindings(aNo[:findings])
        #--> [oneof @ reply] field 'reply' is a closed enumeration (your order is received. |
your order is ready. | we are closed tonight.) and the model answered 'Your order is free,
tonight only.'.
}
```

✓ exécuté : a affiché ce que ce bloc promet

La phrase approuvée est passée (`1`), la phrase inventée a été refusée (`0`), et la raison du tribunal commence par `[oneof @ reply] field 'reply' is a closed enumeration`.

LE MÉDIUM DU RESTAURANT

Le chapitre 1 dit que le propriétaire du restaurant a fait seul sa première démonstration, sur son propre téléphone, avec une application de messagerie comme ligne d'assistance, et le dossier le tient : la liste de vérification de la démonstration du 15 août 2026, et les mesures du 19 août qu'il a envoyées par la même application. Une ligne d'assistance n'est pas un médium. Le chapitre 2 donne déjà son outil comme un tableur. Son propre rapport du 4 juillet 2026 donne comme outil actuel un tableur. Il demande une page web ouverte par un code QR ou par un bouton d'un réseau social. Rien de lourd à télécharger. La ligne sur diverses applications, dont une application de messagerie, est l'étiquette d'une option du formulaire de cadrage, et ce n'est pas ce qu'il a coché. "Simple comme une application de messagerie" est simulé dans le projet, comme le chapitre 28 l'a déjà constaté.

Le restaurant est donc la première forme de la phrase de ce chapitre, et non un exemple du médium de messagerie. Le client garde son médium : un lien qui s'ouvre dans le navigateur qu'il a déjà. Le médium de messagerie, comme thèse, n'est pas

construit. Je raconte cette histoire et ne revendique d'une application de messagerie rien de plus que ce que le dossier tient : une ligne d'assistance, et les photos d'un jour. Le chapitre 1 et cette section disent la même chose.

Voilà la forme, construite et exécutée ; viennent ensuite ce que j'ai décidé et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

Les refus ci-dessous, après le premier, sont ceux du dossier : des décisions d'une session, écrites dans les en-têtes des fichiers de son de la bibliothèque et dans son plan de la voix. Je les rapporte comme tels, et ce ne sont pas des décisions de moi.

La messagerie, la question. Ma demande, telle que mon étude du dossier la rapporte : les conversations ont lieu de façon programmatique par défaut, ou avec une configuration minimale, sans obliger les utilisateurs à changer leurs cultures d'usage. Les propositions pour les surfaces conversationnelles qui la suivent sont celles d'une session, et le dossier ne contient aucune décision de ma part qui les adopte. Je dis *proposé*, pas *décidé*.

Une conversation n'a pas besoin de modèle d'IA. Le plancher est déterministe. Un modèle peut améliorer la fluidité derrière la même surface.

La voix n'est pas un paradigme. Le plan du dossier dit hors de portée jusqu'à demande écrite : toute la gestion du dialogue, les mots d'éveil, l'analyse d'intentions et l'interruption par l'utilisateur.

Des commandes, pas de la transcription : un refus mesuré. La barrière a été fixée avant les chiffres : la dictée libre est livrée à quatre-vingts pour cent d'exactitude et 0,60 de confiance moyenne. L'exécution a donné 66,7 pour cent et 0,600, ce qui a échoué. La grammaire fermée a donné cent pour cent et 0,898, ce qui a réussi. Cette face n'a donc pas de `Transcribe()`, et n'en aura jamais tant qu'un meilleur module de reconnaissance n'existera pas.

Refuser, ne jamais substituer. Parler français à un opérateur qui a demandé l'anglais est pire que de ne rien dire.

Pas de sens inféré. L'artisan du logiciel déclare toujours le sens. Une sixième valeur serait un amendement constitutionnel portant les habits d'une bibliothèque.

Un puits prouve ce que vous auriez envoyé. La limite honnête du port SMS : le filtrage des opérateurs de téléphonie, les règles d'identifiant d'expéditeur, les particularités d'alphabet par pays, et les accusés de réception asynchrones. Rien de cela n'est modélisé ici.

L'écran acquitte, et le son n'est pas l'acquittement.

La section suivante nomme les gardes qui se tiennent derrière ces exécutions.

5. LE TRIBUNAL

`sms_port_narrated` prouve la loi des segments, avec ses frontières à 160 et 161, 306 et 307, la falaise des émojis de 100 à 102 unités, 71 lettres arabes faisant deux segments, et les neuf caractères d'extension. Elle prouve l'E.164, le puits et sa facture, la panne contre le rejet, et le refus en production. Je l'ai exécutée une fois, et 53 assertions sur 53 ont passé. `wisecoding_rich_narrated` est celle du chapitre 18 : le cadre, les options numérotées, le refus d'un nombre que personne n'a offert, les points de contrôle avec une durée de vie. `payments_webhooks_narrated` prouve qu'une signature entrante est vérifiée avant d'être crue : non signée, falsifiée, rejouée. `sound_semantics_narrated` prouve que les cinq valeurs sont celles de la loi, que quatre sonnent et que `muted` est silencieuse, et qu'une valeur inventée est refusée. `sound_listener_narrated` prouve la grammaire fermée, que l'absence de correspondance est un résultat, et que la confiance est un appel séparé ; je l'ai exécutée une fois, et 21 ont passé. `sound_convergence_narrated` prouve que la boucle se ferme pour les quatre états, que la langue est négociée par direction, qu'une sixième valeur est refusée à la déclaration, et que la phrase d'un inconnu ne donne aucun sens.

Une réserve pour la ligne du tribunal. Les gardes du son ont besoin de la bibliothèque sonore du moteur, et pour la boucle, d'une voix et d'un module de reconnaissance. Sur une machine sans eux, les gardes de voix impriment "skipped" et passent. Un succès là ne prouve rien sur la parole. Les gardes pour le SMS et pour la conversation n'ont pas cette dépendance.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`sms_port_narrated` 53 assertions
base/test/system/sms_port_narrated.ring

`wisecoding_rich_narrated` 52 assertions
base/test/conversation/wisecoding_rich_narrated.ring

`sound_convergence_narrated` 26 assertions
base/test/sound/sound_convergence_narrated.ring

`sound_listener_narrated` 21 assertions
base/test/sound/sound_listener_narrated.ring

`sound_semantics_narrated` 41 assertions
base/test/sound/sound_semantics_narrated.ring

`payments_webhooks_narrated` 51 assertions
base/test/system/payments_webhooks_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Mon étude du dossier a compté les unités constitutives de la messagerie, dix-huit en tout, et ce qui manque. Sa phrase dit que le mot désignant une application de messagerie n'apparaît dans aucun fichier de code ni aucun document de conception de la bibliothèque, et l'USSD dans aucun. Je l'ai remesurée à ce commit et confirmée. Il n'y a aucun résultat dans le code ni dans la conception de base, core ou max. Les correspondances dans le reste de l'instantané sont des fichiers de données de listes de mots et de test, et l'USSD n'en a aucune en texte.

Les cadres sont en tête pour la compréhension et pour les connecteurs. Le chapitre 18 énonce le côté de Softanza à partir du code : la conversation demande un champ à la fois, sur un graphe que l'utilisateur doit d'abord avoir déclaré. Elle ne comprend pas une réponse ouverte.

Le terrain est en tête pour la couverture des canaux et pour la vérité de la livraison, les accusés de réception et les gabarits approuvés par le fournisseur. Le port SMS le dit lui-même : un puits prouve ce que vous auriez envoyé, pas qu'un opérateur l'accepterait, et l'adaptateur réel dépend de l'infrastructure. Aucun client réel de SMS n'est livré.

Le plan de la voix : une déclaration est portable, et une voix ne l'est pas. La voix d'un navigateur n'a pas de tampon, donc la boucle ne se ferme que sur le niveau natif. Cette machine peut parler anglais et ne peut pas l'entendre.

Ce que le terrain a en dictée dans le nuage et en modules de reconnaissance entraînés pour beaucoup de langues, le dossier a choisi de ne pas le livrer. La dictée libre a atteint 66,7 pour cent sur l'audio synthétique propre du volet lui-même.

Ce que le dossier affirme à la place : le dialogue est un rendu d'un monde déclaré jugé par un tribunal, déterministe par défaut. Le médium est un port avec un bac à sable, donc la conversation et sa facture sont répétées avant qu'aucun compte existe.

Le terrain est en tête sur certains points, et la limite vient ensuite, imprimée entière.

7. LA LIMITE

CONSTRUIT — la conversation gouvernée avec sa question comme donnée ; le port SMS et son bac à sable, avec la facture par segment dans deux alphabets ; un vérificateur du corps d'un message entrant ; cinq valeurs sémantiques rendues en couleur, en earcon et en phrase ; une voix qui parle, et qui entend une grammaire fermée ; un ensemble fermé de phrases qu'une entreprise accepte de dire ; un message de trois champs entre identités d'une même plateforme

CONÇU — le service du socle commun pour la messagerie ; des surfaces conversationnelles pour une application de messagerie, le SMS, l'USSD et

la voix, avec une matrice de satisfaisabilité comme tribunal ; des gabarits comme grammaire

ABSENT — une application de messagerie, l'USSD ou une ligne téléphonique, sous quelque forme que ce soit ; une projection conversationnelle d'un monde ; un canal, une session, un gabarit, le consentement ; la question dans une autre langue que l'anglais ; une voix pour le haoussa, le zarma ou le fulfulde ; les vingt-deux verbes comme liste que la bibliothèque détient ; une réponse contenant un espace

Mon étude du dossier contient deux erreurs que les exécutions de ce chapitre corrigent. La première : elle dit que la parole vers texte manque. À ce commit elle existe, par décision, pour les commandes et non pour la dictée. La seconde : elle dit qu'aucun message n'existe. Il existe un message de trois champs, entre identités d'une même plateforme, et je le montre :

HARO

```
oDb = new stzDatabase(":memory:")
StzPlatformQ("neighbourhood") {
        OpenCommonsOn(oDb)
        PostMessage("customer", "kitchen", "one brik, no egg")
        ? @@(Inbox("kitchen"))
        #--> [ [ "customer", "one brik, no egg" ] ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

C'est un expéditeur, un destinataire, un corps et une heure dans une table SQL. Il n'a ni médium, ni statut, ni parties. C'est le message que mon étude a trouvé absent, dans sa plus petite forme.

Ce qui est construit : la conversation gouvernée ; le port SMS et son bac à sable ; un vérificateur de signature de corps avec protection contre le rejeu, câblé à aucun fournisseur de messagerie ; la couche sémantique de cinq valeurs rendues en couleur, en earcon et en phrase. Construits aussi : la voix ; un message minimal dans le socle commun ; un ensemble fermé de phrases avec un refus ; et les intentions et portées déclarées du monde.

Ce qui est conçu : `Provides(:Messaging)` comme service du socle commun de la constellation. Un seul fichier de conception le dit, en deux lignes : un schéma qui range la messagerie parmi les communs, et une esquisse qui écrit `Provides(:Messaging)`. Le mot `Provides` n'est dans aucun fichier de code du répertoire de l'application. Le runtime du socle commun comme pendant opérationnel des fournitures et partages déclarés. Les surfaces conversationnelles pour une application de messagerie, le SMS, l'USSD, la voix et le chat web, la matrice de satisfaisabilité comme tribunal, les gabarits comme grammaire, un canal dans le profil de plateforme : ce sont des propositions de mon étude du dossier, écrites par une session, et non la conception de la bibliothèque. Une analyse parlée et un transport parlé : des démonstrations et des compositions, et non une unité.

Ce qui est absent :

- Des ports pour une application de messagerie, Telegram, l'USSD, et l'appel téléphonique : zéro résultat. Une session USSD n'a aucune forme dans la bibliothèque.
- Une projection conversationnelle d'un monde. La seule projection est `Generate`, qui connaît trois surfaces et en refuse une quatrième. Aucun shell ne rend une question ou un flux.
- Les unités de messagerie : canal, session avec une fenêtre ou un délai, gabarit, consentement. Aucune unité de consentement n'existe pour la messagerie. Le mot *consentement* apparaît dans l'instantané pour la frontière du microphone, dans des commentaires, et dans deux documents de conception en d'autres sens.
- Les vingt-deux verbes comme liste de la bibliothèque. `stzApp` en tient cinq comme cinq méthodes d'intention d'écran, et l'assistant derrière elles ne contrôle pas le mot qu'on lui donne. Les dix-sept autres sont dans la constitution et non dans la bibliothèque. La phrase-pont est appliquée à la main aujourd'hui.
- La question dans une autre langue que l'anglais. `AskIn` est en anglais, et le tuteur montre le motif en quatre langues. Dans ma recherche à ce commit, `NaturallyIn` demandé pour le zarma a levé l'opaque "Unsupported object type!" et non "no pack for this language" ; aucun bloc de ce livre n'enregistre cet appel.

- Une voix pour les langues que les conceptions de produit nomment, le zarma, le haoussa et le fulfulde. Aucune ici ; la bibliothèque refuserait.
- **Une réponse contenant un espace, venant d'un téléphone.** Dans ma recherche à ce commit, j'ai exécuté `ReplyIn("c", "grand couscous")` et il a levé une erreur, avec une variable non initialisée. Avec un trait d'union il a été admis. Aucun bloc de ce livre n'enregistre l'un ou l'autre résultat, comme le chapitre 18 le dit de la même faute. C'est la limite connue du chapitre 18, et elle est décisive ici, parce que les gens tapent des espaces dans un SMS. La conversation ci-dessus ne fonctionne que parce que les réponses sont des chiffres et des plats d'un seul mot.
- Des adaptateurs réels pour le SMS, pour une application de messagerie et pour une ligne téléphonique ; des accusés de réception de livraison ; des règles d'identifiant d'expéditeur.
- Rien dans le dossier ne tourne sur un vrai téléphone comme conversation.

`On(:whatsapp)`, `On(:ussd)`, `On(:voice)` et la matrice de satisfaisabilité sont ce que mon étude du dossier a proposé, et je ne les présente pas comme ce que je compte construire. Je n'ai trouvé dans le dossier aucune décision de ma part de les construire, et la limite est la liste ci-dessus, où aucun d'eux n'existe.

Voilà la limite, imprimée entière ; les six portes suivent.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	La question revient comme donnée. Vous écrivez le rendu et le lecteur de votre médium en une douzaine de lignes, et chaque réponse passe par la même porte quel que soit ce qui l'a portée.
L'analyste	Le but est le schéma de la conversation. La transcription dit quel message a produit quel fait. Le bac à sable compte la facture avant qu'un fournisseur existe.
Le concepteur	Votre utilisateur garde le téléphone qu'il a : une ligne de texte avec des numéros, ou une phrase parlée. Les verbes de la constitution sont la grammaire que vous rendez, et un verbe que le médium ne peut pas rendre devrait être une cellule rouge, et non une omission silencieuse. C'est conçu et non construit, puisque la matrice n'existe pas.
Le décideur	Une conversation par SMS peut être chiffrée dans un test : deux noms de plats en arabe ont fait passer une question de 93 caractères d'un segment à deux. La question à poser à un fournisseur, qu'est-ce qui refuse une phrase que votre entreprise n'a pas approuvée, a ici une réponse : un ensemble fermé.
Le formateur	Le tuteur dit déjà l'étape manquante comme une question en quatre langues, dans des gabarits qui ne portent aucun code. Le dossier ne nomme pas d'apprenant sur un téléphone simple, et rien n'en sert un pour l'instant : il n'y a ni USSD ni application de messagerie.
L'agent	Vous pouvez proposer une réponse, et seule une phrase de l'ensemble fermé passe. Une phrase inventée est refusée avec sa raison. Et vous ne pouvez pas donner un sens à une phrase : l'artisan du logiciel le déclare.

9. EXERCICE

Prenez une question que votre propre monde pose. Le monde à une relation de l'exercice du chapitre 18 fera l'affaire. Avant d'exécuter quoi que ce soit, écrivez sous chaque ligne la sortie que vous attendez :

1. le nombre de segments de votre rendu de la question, avec des noms en alphabet latin ;
2. ses segments si une option est un mot en écriture arabe ;
3. ce qui se passe quand la réponse du client est un chiffre qui n'a pas été offert ;

4. ce qui se passe quand la réponse est "yes please".

HARO

```
? StzSmsSegments(YourRendering)[:segments]
#--> your prediction
? StzSmsSegments(YourRenderingWithOneArabicWord)[:segments]
#--> your prediction
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Exécutez-le, avec le bac à sable et jamais un vrai envoi, pour que l'exercice ne touche à rien. Si une promesse casse, ne la changez pas : expliquez d'abord la sortie.

Je vous laisse celui-ci. La quatrième prédiction est un défaut que je n'ai pas corrigé, et la section 7 le liste en gras : une réponse contenant un espace, limite connue du chapitre 18. Reprenez le bloc de la section 3 qui envoie une réponse à une question sans option sur la table, remplacez la réponse "2" par un texte contenant un espace, et exécutez-le :

HARO

```
oKT = new stzKnowledgeGraph("trap")
oKT.Know("order1", "order")
oKT.AddConversationQ("c").SetGoal(StzGoalQ().RequireEach("order", "dish"))
oKT.AskIn("c")
? @@(oKT.ReplyIn("c", "grand couscous")[:admitted])
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Regardez ce que fait l'appel, et ne le réparez pas. La section 7 dit pourquoi cela compte sur un téléphone. Quand vous aurez expliqué la sortie, vous saurez pourquoi la conversation ci-dessus ne fonctionne que parce que ses réponses sont des chiffres et des plats d'un seul mot.

Le chapitre suivant est l'environnement. Il déclare l'interface une fois, et chaque shell en est une projection ; une projection, comme un médium, peut ajouter des rendus, jamais des verbes.

OÙ SONT LES PREUVES

NARRATION	stz-service-virtualization-code-first-subscribe-later.md
NARRATION	stznatural-narration.md
NARRATION	stzApp-building-your-first-app-narration.md
SUR LE SITE	zui
DÉCISIONS DE L'ÉTUDE	12.15, 12.23, 12.24, 12.25, 12.26, 12.27

PARTIE V · CHAPITRE 26

L'environnement

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 10 bloc(s) sur 10 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends la forme que les environnements du terrain ont fini par prendre, une barre d'outils, une barre latérale, un éditeur et une barre d'état, et je lui oppose un motif en cinq marques. J'exécute les pièces que la bibliothèque a sous ces marques : un shell déclaré comme texte et jugé, deux projections, un ruban dans l'ordre d'une vie, un rail qui propose, une barre d'état qui dit la vérité. Je dis ce que j'ai décidé, porter les deux shells, et ce que j'ai refusé, dont un drapeau qui désactive l'arbre d'accessibilité. Et je dis où cela en est : à ce commit, un IDE de cette famille est absent, et les pièces qui le portent tournent. Le principe tient en une phrase : déclarer l'interface une fois, et chaque shell en est une projection qui peut ajouter des rendus, jamais des verbes.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Les environnements du terrain ont fini par prendre une forme. Une barre d'outils et un menu en haut, une barre latérale de fichiers, un éditeur, un terminal en dessous, et une barre d'état en bas. Depuis peu un assistant se tient dans un volet ou un terminal à côté du code, et l'humain approuve ses modifications. Je prends cette forme de la connaissance générale, et non du dossier.

Les études de terrain propres du dossier, datées, sont au nombre de trois. Le bloc-notes livré avec l'ancien langage hôte, mesuré le 12 août 2026 à trente-trois fichiers source et environ quatre mille lignes, sept volets ancrables, une configuration de quinze clés et aucun service de langage. Il existe, il est livré dans la boîte, et il

fonctionne le jour de l'installation. Les éditeurs rapides, dans la spécification d'ergonomie de mars 2026, qui fixe pour son propre produit : de l'entrée à l'écran en moins de seize millisecondes sur tout matériel moderne, sans exception. Elle veut un chrome minimal, pas de barre d'état encombrée, pas de forêts de barres d'outils. Elle demande un essai à blanc avant un changement structurel, et des inspections qui disent pourquoi et proposent une correction. Et la barre d'état, telle que cette spécification l'écrivait : vingt-quatre pixels de haut, ne portant que l'état essentiel, avec cinq états de construction, dont l'un est *non vérifié*, pour un projet où aucune construction n'a encore tourné. L'état de « pas encore jugé » a son propre mot. Une barre verte par défaut serait le mensonge habituel du terrain.

Là où le terrain est silencieux, le chapitre répond. Le terrain ordonne ses barres d'outils par les fonctionnalités de l'outil, et non par la vie de la chose qu'on fait. Et il laisse la proposition d'un assistant atterrir comme une modification, l'approbation de l'humain étant une habitude ; cela aussi est connaissance générale.

2. LE PREMIER PRINCIPE

Déclarer l'interface une fois. Chaque shell en est une projection, et une projection peut ajouter des rendus, jamais des verbes.

L'en-tête de la bibliothèque pour son interface rédigée le dit : le fichier est le contrat, et le balisage natif et la page web en sont les projections. Mon étude du dossier le lit comme une déclaration, deux projections, prouvées d'accord, reposant sur ma décision du 7 octobre 2026 : porter les deux. La phrase compagnon est celle de la constitution de l'interface, sur le site : un nouveau médium ajoute des rendus, jamais des verbes.

3. LA FORME SOFTANZA

D'abord l'étape, avant les blocs. À ce commit, **un IDE de cette famille est absent**. Aucune classe de la bibliothèque n'est encore nommée pour un ruban, un rail, une colonne ou une barre d'état. Mon étude du dossier l'a mis dans une ligne d'étape proposée pour le site : aucun IDE de cette famille n'est livré aujourd'hui ; un

shell l'est, pour le volet sonore. La fondation qui ferait du suivant une projection est spécifiée et pas encore construite. J'ajoute une chose : la bibliothèque génère une surface de navigateur, celle de l'émulateur, et je la montre plus bas avec une perte.

La bibliothèque a les pièces sous chacune des cinq marques du motif. Les cinq sont un shell, deux projections, un ruban de facettes ordonné par la vie de la solution, un rail qui propose et une barre d'état qui dit la vérité. Elles tournent, et je les prends dans l'ordre.

UN SHELL, DÉCLARÉ COMME TEXTE, JUGÉ PAR UN TRIBUNAL

Le chrome est déclaré comme texte et jugé avant d'être montré. Le ruban est généré à partir de données, ici une simple liste ; la facette comme objet de bibliothèque est conçue et non construite. Les quatre noms de facettes sont quatre des cinq phases du corpus de conception du dossier : définir, architecturer, construire, gouverner, livrer. J'en ai pris quatre pour garder le bloc court. Chaque déclaration doit porter sa justification, et une déclaration sans elle est refusée. Quel élément ne sert aucun verbe ? Exécutez-le :

HARO

```
aFacets = [ "Define", "Build", "Govern", "Ship" ]
cTabs = ""
cIds = ""
for cF in aFacets
        cId = "tab_" + lower(cF)
        cIds += cId + ", "
        cTabs += 'DEFINE TEXT ' + cId + ' ( CONTENT "' + cF + '", ROLE tab, SATISFIES select )
RATIONALE "A facet."' + nl
next
cShell = 'DEFINE PANEL shell ( SIZE [640, 360], DIRECTION column, FONT "app", SATISFIES locate,
CHILDREN [ribbon, work, status] ) RATIONALE "One shell, three bands."' + nl +
        'DEFINE BOX ribbon ( HEIGHT 32, DIRECTION row, ROLE tablist, SATISFIES discover, CHILDREN
[' + cIds + '] ) RATIONALE "Facets, in the order of a life."' + nl + cTabs +
        'DEFINE BOX work ( HEIGHT fill, DIRECTION row, CHILDREN [page, rail] ) RATIONALE "The
workspace and its rail."' + nl +
        'DEFINE TEXT page ( CONTENT "the work", SATISFIES read ) RATIONALE "What is being made."'
+ nl +
        'DEFINE BOX rail ( WIDTH 160, ROLE group, SATISFIES preview, CHILDREN [advice] )
RATIONALE "What the court proposes."' + nl +
        'DEFINE TEXT advice ( CONTENT "no proposal", SATISFIES understand ) RATIONALE "The rail,
empty."' + nl +
        'DEFINE BOX status ( HEIGHT 24, ROLE status, SATISFIES understand, CHILDREN [state] )
RATIONALE "The truth of the build."' + nl +
```

```
        'DEFINE TEXT state ( CONTENT "clean", SATISFIES understand ) RATIONALE "The court
verdict."'
oShell = new stzUiDocument(cShell)
oShell {
        ? IsClean()
        #--> 1
        ? len(Declarations())
        #--> 12
        ? @@( Orphans() )
        #--> [ "work" ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

`Orphans()` répond `[ "work" ]`, et `IsClean()` répond `1` sur les `12` déclarations.

La boucle de génération est celle du livre, et les rôles viennent de la liste fermée de la bibliothèque : `tablist`, `tab`, `group` et `status` y figurent.

LES VINGT-DEUX VERBES

Chaque élément doit se rattacher à au moins un verbe, et un mot qui n'est pas l'un des vingt-deux est trouvé. Il y a une division du travail. La bibliothèque enregistre l'affirmation, avec `SATISFIES`, et trouve les éléments non rattachés, avec `Orphans()`. Elle ne détient pas la liste et ne refuse pas une intention inconnue : la garde affirme qu'une intention que personne n'a définie n'est pas une erreur. Donc la liste ci-dessous est saisie d'après la constitution, qui est dans son propre dépôt, et le livre la détient. Le tribunal refuse-t-il une intention qui n'est pas dans la liste ? Exécutez-le :

HARO

```
cShell = StzReplace(cShell, "DEFINE BOX work ( HEIGHT fill,", "DEFINE BOX work ( SATISFIES focus,
HEIGHT fill,")
oShell = new stzUiDocument(cShell)
? @@( oShell.Orphans() )
#--> [ ]
aVerbs = [ "discover", "understand", "locate", "focus", "filter", "compare", "read", "scan",
"scroll", "zoom", "select", "highlight", "preview", "mark", "act", "confirm", "cancel", "undo",
"pause", "resume", "retry", "exit" ]
? len(aVerbs)
#--> 22
aStrangers = []
```

```
for cIntent in oShell.SatisfiedIntents()
        if not StzContains(aVerbs, cIntent)
                aStrangers + cIntent
        ok
next
? @@( aStrangers )
#--> [ ]
oOdd = new stzUiDocument(StzReplace(cShell, "SATISFIES preview", "SATISFIES transfer_funds"))
? oOdd.IsClean()
#--> 1
aStrangers = []
for cIntent in oOdd.SatisfiedIntents()
        if not StzContains(aVerbs, cIntent)
                aStrangers + cIntent
        ok
next
? @@( aStrangers )
#--> [ "transfer_funds" ]
```

✓ exécuté : a affiché ce que ce bloc promet

Avec `SATISFIES focus` sur `work`, `Orphans()` répond `[ ]`, et la boucle sur la liste du livre aussi. Interrogé sur `transfer_funds`, `IsClean()` répond `1` ! La boucle sur la liste du livre répond `[ "transfer_funds" ]`.

`work` était une pure boîte de mise en page, et le tribunal compte aussi les conteneurs. Savoir si un conteneur doit être exempté des verbes appartient à la constitution, et non à la bibliothèque. Les familles, d'après la constitution : orientation, attention, information, sélection, action, et continuité. Et l'interprétation que je donne : la bibliothèque trouve l'étranger, et une liste que la bibliothèque ne détient pas ne peut pas le refuser. Le chapitre 25 a dit que la phrase-pont est appliquée à la main aujourd'hui. Je raconte ce retrait comme le refus qui est au centre de ce chapitre, dans la section 4 : la bibliothèque ne possède pas les verbes, la constitution les possède. Le texte de conception de la bibliothèque en dit encore vingt-quatre ; ce n'est pas la liste. Les vingt-deux sont imprimés dans le bloc ci-dessus, saisis d'après la constitution comme l'annexe C les imprime.

DEUX PROJECTIONS

Le même texte donne le balisage natif et la page web. Chaque élément déclaré est dans les deux, et les deux écrivent une propriété différemment exactement là où le profil enregistre une divergence. Combien de noms sont dans les deux ? Comptez-les

:

HARO

```
cRml = oShell.ToRml()
cHtml = oShell.ToHtml()
nBoth = 0
for cName in oShell.Names()
        if StzContains(cRml, 'id="' + cName + '"') and StzContains(cHtml, 'id="' + cName + '"')
                nBoth++
        ok
next
? nBoth
#--> 11
? len(oShell.Names()) - 1
#--> 11
? StzContains(cRml, "--rmlui-direction")
#--> 1
? StzContains(cHtml, "--rmlui-direction")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Onze éléments et non douze : le panneau est la page elle-même, et non un `div`. Les deux dernières lignes montrent où les deux diffèrent. Le balisage natif contient `--rmlui-direction` (`1`), et la page web non (`0`). Le niveau natif met en page par la bibliothèque de mise en page du moteur, embarquée, et dessine par le volet graphique. Je n'ai pas ouvert de fenêtre.

LE SHELL DU NAVIGATEUR

Le shell du navigateur est une page que le langage sert, sur la boucle locale. Une requête, un processus, aucun réseau au-delà de 127.0.0.1. Un second réacteur joue le client, donc c'est hors ligne par construction. Le gestionnaire lit une globale, parce que les lambdas de l'ancien langage hôte ne capturent pas les locales. Où vit la page ? Demandez au serveur :

HARO

```
$cShellPage = cHtml
oSrv = new stzAppServer()
oSrv.Get_("/", func oReq, oResp { oResp.Html($cShellPage) })
oSrv.Start(0, "127.0.0.1")
```

```
oClient = new stzReactor()
cCRLF = char(13) + char(10)
nJob = oClient.SubmitTcp("127.0.0.1", oSrv.Port(), "GET / HTTP/1.1" + cCRLF + "Host: local" +
cCRLF + "Connection: close" + cCRLF + cCRLF)
? oSrv.ServeOne(3000)
#--> 1
cAnswer = oClient.AwaitTcp(nJob, 5000)
? StzContains(cAnswer, "200 OK")
#--> 1
? StzContains(cAnswer, 'id="status"')
#--> 1
? oSrv.Host()
#--> 127.0.0.1
oSrv.Stop()
```

✓ exécuté : a affiché ce que ce bloc promet

`Host()` répond `127.0.0.1`. Le coût de cette forme est nommé dans le dossier. Un port lié localement est une surface et doit être traité comme tel. Selon les mots de la charte, l'intention de conception a été mesurée pour un studio de son et non pour un IDE. Le backend de la bibliothèque refuse déjà de quitter la boucle locale sans authentification, au moment de lier le port ; le serveur nu ci-dessus ne vérifie rien. Que fait le backend de `0.0.0.0` ? Essayez :

HARO

```
oT = new stzAppTopology("shell")
oT.AddDataset("notes", [ [ "first" ] ])
oT.SetDatasetColumns("notes", [ "text" ])
oBackend = new stzAppBackend("shell", oT)
bRaised = FALSE
try
        oBackend.StartOn("0.0.0.0", 0)
catch
        bRaised = TRUE
done
? bRaised
#--> 1
? oBackend.IsLive()
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Il lève une erreur ! `bRaised` répond `1`, et `IsLive()` répond `0`.

UNE BARRE D'ÉTAT QUI DIT LA VÉRITÉ

Une cellule d'état est une fonction du verdict du tribunal, donc elle ne peut pas dire *propre* d'un document que le tribunal a refusé. La fonction ci-dessous est celle du livre, au-dessus de `IsClean()` de la bibliothèque et de ses diagnostics. La classe de cellule d'état n'est pas encore dans la bibliothèque, et la garde qui interdit qu'une cellule montre autre chose que ce que le tribunal a émis est conçue. La spécification nomme aussi l'état avant toute construction, *non vérifié*, et ma fonction ne le porte pas. Que dit la cellule pour un enfant déclaré nulle part, et pour un rôle qui n'est pas dans la liste ? Exécutez-le :

HARO

```
? ShellStatus(cShell)
#--> clean
? ShellStatus(StzReplace(cShell, "[page, rail]", "[page, ghost]"))
#--> refused: DANGLING_REFERENCE
? ShellStatus(StzReplace(cShell, "ROLE status", "ROLE sttus"))
#--> refused: UNKNOWN_ROLE

func ShellStatus(cText)
        oD = new stzUiDocument(cText)
        if oD.IsClean()
                return "clean"
        ok
        return "refused: " + oD.Diagnostics()[1][:code]
```

✓ exécuté : a affiché ce que ce bloc promet

La cellule répond `refused: DANGLING_REFERENCE` et `refused: UNKNOWN_ROLE`, et *propre* seulement pour le document que le tribunal a gardé.

Et ce que lit un lecteur d'écran est émis depuis la déclaration. Une région déclarée `ROLE status` est émise dans l'arbre d'accessibilité avec sa justification comme description, dans l'ordre de lecture, et aucun drapeau ne désactive l'arbre. La mise en page a besoin d'un fichier de police, et ce bloc utilise celle du système. Que dit l'arbre de la région d'état ? Demandez-lui :

HARO

```
oShell.UseFont("C:/Windows/Fonts/segoeui.ttf")
oP = oShell.ToPanel()
```

```
oT = new stzAccessibilityTree(oShell, oP)
oT {
        ? NodeCount()
        #--> 12
        ? NodeOf("status")[:role]
        #--> status
        ? NodeOf("status")[:description]
        #--> The truth of the build.
        ? @@( ReadingOrder() )
        #--> [ "window", "tablist", 'tab "Define"', 'tab "Build"', 'tab "Govern"', 'tab "Ship"',
"group", 'label "the work"', "group", 'label "no proposal"', "status", 'label "clean"' ]
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'arbre répond `status`, et pour sa description `The truth of the build.`, la justification mot pour mot ! L'en-tête de la bibliothèque dit pourquoi : chaque toolkit garde l'accessibilité derrière un drapeau de performance, et c'est là que les utilisateurs n'obtiennent silencieusement rien. Donc ici il n'y a rien à garder derrière un drapeau et rien à oublier d'activer. Le même fichier dit ce qu'un lecteur d'écran peut manipuler, un agent d'IA peut le manipuler.

UN RAIL QUI PROPOSE

Un rail est une vue sur ce que le monde propose. Une pulsation propose et ne relie jamais. Un acte sur le monde efface une proposition, et la ligne d'état propre du monde compte à partir de la même liste que montre le rail. Que propose le monde, et que reste-t-il après un acte ? Exécutez-le :

HARO

```
StzAppQ("fieldvisits") {
        AddThingQ(:Client) { Has([ :name ]) }
        AddThing(:Visit)
        AddReactionQ(:Client).Unseen(90, :Days) { Propose(:Visit) }
        AddInstance("amadou", :Client)
        AddInstance("binta", :Client)
        Live()
        #--> [fieldvisits] is live -- 2 thing(s), 0 flow(s), 1 reaction(s), 0 goal(s); 2
proposal(s)
        for aP in Proposals()
                ? aP[1] + " " + aP[2] + " " + aP[3] + " " + aP[4]
        next
        #--> propose visit for amadou
        #--> propose visit for binta
```

```
        ? Pulse()
        #--> 0
        ? len(Proposals())
        #--> 2
        Relate("amadou", :visit, "visit-1")
        Pulse()
        for aP in Proposals()
                ? aP[1] + " " + aP[2] + " " + aP[3] + " " + aP[4]
        next
        #--> propose visit for binta
}
```

✓ exécuté : a affiché ce que ce bloc promet

`Live()` imprime sa ligne, `2 proposal(s)` à sa fin, et le juge compare ce qui est imprimé. Avant l'acte, `Pulse()` répond `0` et le rail tient encore 2 propositions. Après l'acte, une ligne reste : `propose visit for binta`. "2 thing(s)" compte les deux choses, Client et Visit, et non les instances. Le chapitre 20 possède la loi sous le rail, la proposition, le plan et le tribunal.

Et un écran est une intention. Un endroit où le monde parle en verbes est celui de ses écrans. Cinq méthodes existent : `ToDiscover`, `ToUnderstand`, `ToFocus`, `ToSelect` et `ToAct`. Un sixième verbe, `ToCompare`, n'existe pas. Qu'est-ce qui répond quand un écran le demande ? Exécutez-le :

HARO

```
oW = StzAppQ("fieldvisits")
oW {
        AddThingQ(:Client) { Has([ :name ]) }
        AddScreenQ(:Home)  { ToDiscover(:Client) }
        AddScreenQ(:File)  { ToUnderstand(:Client) }
        AddScreenQ(:Pick)  { ToSelect(:Client) }
        AddScreenQ(:Edit)  { ToAct(:Client) }
        Explain()
        #--> WORLD fieldvisits   lives in: memory (not persisted)
        #      BEING
        #        client (name)
        #      MET FROM WITHOUT
        #        screen home: discover client
        #        screen file: understand client
        #        screen pick: select client
        #        screen edit: act client
}
try
        oW.AddScreenQ(:Side) { ToCompare(:Client) }
```

```
catch
        ? cCatchError
done
#--> Error (R3) : Calling Function without definition: tocompare
```

✓ exécuté : a affiché ce que ce bloc promet

C'est le runtime qui répond, `Error (R3) : Calling Function without definition: tocompare`, et aucun tribunal : le mot d'intention est stocké comme chaîne, et n'est vérifié contre aucune liste.

Voilà ce qui tourne ; je dis maintenant ce que j'en ai décidé, et ce que j'ai refusé.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé de porter les deux shells. Le 7 octobre 2026 on m'a demandé si la fondation prend le shell natif ou le shell du navigateur, et j'ai répondu : porter les deux. La conséquence, telle que mon étude du dossier l'énonce, est qu'un chrome est déclaré une fois, et que deux shells le projettent.

J'ai décidé que la mécanique est celle de la bibliothèque et que l'IDE est celui du produit. Le document de conception de la bibliothèque : la mécanique du paradigme est plateforme ; le livre, la marque, l'IDE sont produit, et ne fuient jamais. La fondation livre le raffinement comme verbe de bibliothèque disponible à tout programme.

J'ai retiré le moteur d'interaction. Le 26 mai 2026 mon commit a retiré vingt-deux modules inventés par un agent d'IA qui n'étaient pas dans le Softanza d'origine, parmi lesquels le module d'interaction. Je lis la date, le compte et la liste dans l'historique git de la bibliothèque, le commit `b70f3ad91`, et non dans l'arbre épinglé. Son texte de conception, qui dit vingt-quatre verbes et des lois appliquées par le moteur, est toujours dans l'arbre. La liste de la constitution en compte vingt-deux.

J'ai refusé de résoudre des intentions à l'intérieur du langage du panneau. Le panneau ne déclare aucun sens : il n'y a pas de `:Danger` et il ne peut jamais y en avoir. L'emplacement d'intention a été laissé délibérément non résolu. Plus haut, `IsClean()` a répondu `1` pour `transfer_funds` : le panneau ne juge aucun sens.

J'ai refusé de rédiger le balisage, et j'ai refusé un moteur de script. Le balisage natif est émis, jamais rédigé ; il n'y a pas de moteur de script ; flexbox seulement. C'est la raison pour laquelle le chrome est un fichier texte et non une classe.

J'ai refusé un drapeau qui désactive l'arbre d'accessibilité. Le bloc de l'arbre ci-dessus a imprimé `status` et `The truth of the build.` sans qu'aucun drapeau soit à activer.

J'ai admis une lentille qui ne peut pas capturer. La charte du domaine de projets pour ses produits : une vérité, dans des fichiers, et deux lentilles, qui dans la charte sont ses deux produits, l'IDE de la porte d'entrée et le studio commercial, et non deux shells. Ce que la charte a refusé, ce sont les cadres qui capturent le travail. Une lentille qui ne peut pas capturer est admissible. L'environnement est une projection de texte brut, et non un magasin.

Une session a retiré une facette, et je l'ai rétablie. La conception de référence consigne que la facette Connexions était de première classe dans le prototype de juin 2026, et qu'un agent d'IA l'a retirée par un raisonnement circulaire. Le retrait s'est propagé en cascade dans cinq documents. Je l'ai rétablie. C'est à moi de décider du ruban.

J'ai accepté le shell du navigateur avec son coût, non mesuré. La charte de l'IDE de la porte d'entrée a choisi un onglet de navigateur, construit avec le moteur, sans étape d'installation, et a nommé la perte.

Le motif a été appris en six efforts sur deux ans, dont un seul construit, selon le compte de mon étude du dossier.

Vient ensuite le tribunal, avec les gardes derrière ces pièces et celles qui manquent.

5. LE TRIBUNAL

`gui_stzui_narrated` prouve la clôture du tribunal du panneau, une référence pendante, un doublon, une boîte orpheline, un panneau manquant, la justification obligatoire et un aller-retour. Elle prouve, à la scène 13, l'emplacement d'intention : `SATISFIES`, `Orphans()`, et qu'une intention que personne n'a définie n'est pas une erreur. `gui_profile_narrated` prouve que le profil est de la donnée, avec trois divergences enregistrées. Elle prouve qu'un document a deux projections et que chaque élément déclaré est dans les deux, et que la fixture émet une page qui se vérifie elle-même. La moitié navigateur de cette garde est une procédure à reproduire. `gui_tier_agreement_narrated` prouve qu'une même liste d'affichage rendue en vectoriel et en pixels s'accorde au pixel près. Ce n'est pas du HTML contre

du natif. `gui_accessibility_narrated` prouve l'arbre émis depuis la déclaration. `05_stzapp_live_narrated` prouve qu'un monde vivant propose et qu'agir efface. `emulator_narrated` prouve la surface générée de l'émulateur.

Trois gardes sont à venir : une qui juge un shell contre les vingt-deux verbes, une qui vérifie qu'une cellule d'état ne montre que ce qu'un tribunal a émis, et une qui dit que les deux projections d'un chrome s'accordent dans le navigateur. Mon étude du dossier les range parmi les absentes, et l'arbre à ce commit est d'accord.

La vérification des deux projections peut échouer, et elle a échoué. La fixture de la bibliothèque est une page et non une assertion : elle porte la projection web et les boîtes que la mise en page native a produites. Un script les compare et imprime une bande de verdict. Un exécuteur ne peut pas la juger, donc je l'ai lue à la main le 8 octobre 2026. Sur le chrome du premier bloc, onze boîtes avec du texte dans chaque région, la bande disait "AGREE on every declared box of 11". Sa note disait que quatre boîtes de largeur automatique diffèrent, ce qui est deux modeleurs de texte et non le profil. Sur le même chrome à trois bandes avec des régions vides, la bande disait "DISAGREE on 1 declared boxes of 3". La cause est dans la projection web : une boîte vide est émise comme `<div id="body"/>`. Un analyseur HTML la lit comme une balise ouvrante, si bien que les frères qui suivent s'y imbriquent. Le niveau natif n'est pas touché. La fixture l'attrape ! Les gardes de la bibliothèque pas encore, puisque leur propre vitrine n'a pas de boîte vide : en ajouter une est l'étape suivante. Un espace de travail vide est le premier état du motif. Donc la bande dit la vérité dans les deux sens, ce qui est la cinquième marque du chapitre en une seule image, et la projection web a un cas à corriger sur cet état exact.

Du tribunal, je reviens au terrain, et à l'endroit où il mène.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`gui_stzui_narrated` 58 assertions
base/test/gui/gui_stzui_narrated.ring

`gui_profile_narrated` 35 assertions
base/test/gui/gui_profile_narrated.ring

`gui_tier_agreement_narrated` 24 assertions
base/test/gui/gui_tier_agreement_narrated.ring

`gui_accessibility_narrated` 37 assertions
base/test/gui/gui_accessibility_narrated.ring

`05_stzapp_live_narrated` 11 assertions
base/test/app/05_stzapp_live_narrated.ring

`emulator_narrated` 39 assertions
base/test/system/emulator_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Les éditeurs rapides fixent une barre que le dossier écrit comme loi pour son propre produit : de l'entrée à l'écran en moins de seize millisecondes, un démarrage à froid en moins de quatre cents, sans exception. Le dossier ne l'a pas encore mesurée sur l'un ou l'autre shell : un chiffre de latence est absent, et c'est l'étape suivante.

Un changement de texte invalide tout le document. Une valeur liée qui change refait la mise en forme de chaque chaîne, là où une couleur n'en refait qu'une. Les éditeurs du terrain appliquent des patchs, et le terrain mène ici aujourd'hui ; c'est de la connaissance générale.

Le navigateur coûte ce qu'il coûte : l'accès aux fichiers par le modèle du navigateur, un port à garder, et une latence non mesurée. Le shell natif coûte un système de fenêtres, et un GPU pour le chemin raster.

Le terrain a un service de langage, comme la connaissance générale le tient, et le dossier a un contrat ; le service est l'étape suivante. Le graphe de monde, `stzWorldGraph`, est le modèle de l'explorateur : il lit le fichier de monde que le service émettrait, et il n'est chargé par aucun fichier d'entrée : un bloc qui l'appelle sans un chargement écrit à la main échoue.

L'éditeur rapide du terrain traite la barre d'état comme de l'encombrement. Le motif en garde une et lui demande d'être vraie. C'est un pari, et je le tiens pour une vue, non pour un résultat ; le corpus propre du dossier plaide pour le côté du terrain dans le même souffle.

Reste la limite.

7. LA LIMITE

CONSTRUIT — l'interface comme texte avec un tribunal qui trouve l'élément ne servant aucun verbe ; deux projections de celle-ci, native et web, avec une table enregistrée des divergences et une page qui peut dire DISAGREE ; l'arbre d'accessibilité émis depuis la déclaration ; une page que le langage sert sur la boucle locale ; un monde qui propose et dont les propositions s'effacent quand on agit dessus ; des écrans déclarés comme intentions ; une surface de navigateur générée

CONÇU — le kit de shells avec deux shells ; le chrome comme genres fermés (colonne, ruban, rail, barre d'état), aucun encore classe de la bibliothèque ; la facette comme donnée ; un rail lié au focus ; des cellules d'état lues dans les tribunaux, avec une garde qui interdit qu'une cellule montre un état que le tribunal n'a pas émis

ABSENT — un IDE de cette famille ; une liste des vingt-deux verbes dans la bibliothèque et un refus d'un verbe inconnu ; une garde qui juge un shell contre les verbes ; un éditeur ; une latence mesurée

La projection web ne porte pas encore de rôles, d'étiquettes ni d'intentions. `ToHtml` n'émet qu'un identifiant, un style et le texte. L'arbre qui porte les rôles est un artefact séparé, comme le bloc ci-dessus le montre, donc un shell de navigateur devrait déduire à partir de `div` nus ce que le shell natif émet. Le plan lui-même dit que la bibliothèque d'accessibilité qu'il adopte n'a pas d'adaptateur web.

Le centre de contrôle de la bibliothèque elle-même imprime des constantes là où le motif demande des verdicts. L'émulateur génère une surface de navigateur, une grille de parties avec des pastilles d'état et des badges de fidélité, une fenêtre d'appareil et une console pour chaque partie. Ses mots d'état sont des constantes de la classe d'appareil et du gabarit, et ils ne sont pas mesurés. Voici un plan avec une partie firmware et une capacité GPIO, et aucune pompe nulle part. Que dit sa page ? Construisez-la et lisez-la :

HARO

```
oPlan = new stzDelivery("greenhouse")
oPlan.AddFirmware(:node, :ESP32)
oPlan.NeedsIn(:node, [ :GPIO ])
oEmu = new stzEmulator(oPlan)
oEmu {
        CompileEngineQ(FALSE)
        SetOutDirQ("_emu")
        Build()
}
cH = read("_emu/index.html")
? StzContains(cH, "needs a look")
#--> 1
? StzContains(cH, "2 approximated")
#--> 1
? StzContains(cH, "parts healthy")
#--> 1
? StzContains(cH, "Ready to ship.")
#--> 1
StzDirDeleteAll("_emu")
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

La partie reçoit "2 approximated" et, dans le détail de la grille, un texte sur la temporisation de la pompe et le bruit des capteurs : le texte de la démonstration de serre, renvoyé pour n'importe quel microcontrôleur. L'en-tête dit "parts healthy" et

la bannière dit "Ready to ship." pour le même plan, à partir de littéraux ! Un plan avec seulement un backend imprime aussi "parts healthy" et "Ready to ship.", et ni "needs a look" ni "2 approximated" (lu dans la source, non exécuté). "Deploy to production" est une alerte. La moitié véridique est réelle : la fidélité est imprimée, et la console refuse un verbe que le sous-ensemble du moteur de la partie n'a pas. C'est de la narration dans la bibliothèque, que je n'ai pas exécutée.

Voilà la perte contre la page de la bibliothèque elle-même. Elle se tient à côté de « une barre d'état qui dit la vérité » de la section 3 : le motif demande qu'une cellule d'état soit une fonction de ce qu'un tribunal a émis. Et la page propre de la bibliothèque imprime ses mots à partir de littéraux. La demande de dériver ces mots du plan est ouverte, et c'est l'étape suivante.

`ToPanel` a besoin d'un fichier de police et lève une erreur sans lui. L'arbre d'accessibilité et la fixture ont besoin d'un panneau mis en page, donc les blocs qui les utilisent dépendent d'une police système.

`Orphans()` compte les conteneurs. Savoir si une boîte de mise en page seule doit servir un verbe est une règle que la constitution doit énoncer.

Ce qui est construit : l'interface comme texte avec un tribunal ; quatre genres fermés, une liste fermée de dix-neuf rôles, un ensemble fermé de champs, une justification obligatoire, un aller-retour jusqu'à un point fixe. Deux projections, avec une table enregistrée de trois divergences et une page qui peut dire DISAGREE. L'arbre d'accessibilité, et un pont qui atteint l'interface d'un vrai lecteur d'écran sous Windows, lu et non exécuté par moi. Une valeur déclarée change et l'écran suit, avec le focus conservé. Une page hébergée sur la boucle locale, et un backend qui refuse de quitter la boucle locale sans signature. Un monde qui propose, des écrans déclarés comme intentions, une surface de navigateur générée. Et un shell binaire, le banc d'écoute du volet sonore, en environ sept cents lignes de Zig, que j'ai lues et pas construites.

Ce qui est conçu : le kit de shells avec deux shells ; le chrome comme genres fermés nommés pour la colonne, le ruban, le rail et la barre d'état, aucun encore classe de la bibliothèque. La facette comme donnée, avec son groupe, sa place dans la vie, ses

genres d'artefacts, son tribunal et son profil. Le rail lié au focus, et les cellules de gouvernance lues dans les tribunaux, avec une garde qui interdit qu'une cellule montre un état que le tribunal n'a pas émis.

Ce qui est absent : un IDE de cette famille. Une liste des vingt-deux verbes dans la bibliothèque, et tout refus d'un verbe inconnu : `transfer_funds` passe le tribunal. Une garde qui juge un shell contre les verbes. Un éditeur. Une latence mesurée. Je raconte le motif comme conçu, avec les pièces qui tournent comme preuve : le tribunal du chrome, les deux projections, l'arbre d'accessibilité, la page hébergée sur la boucle locale. Aucun IDE de cette famille n'est livré aujourd'hui.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous écrivez le shell comme texte, et un tribunal vous dit l'unique boîte qui ne sert aucun verbe avant que quoi que ce soit soit dessiné. Le même fichier est une fenêtre native et une page, et la bibliothèque a une page qui dit quand elles divergent.
L'analyste	Le ruban est l'ordre de la vie de votre solution, et non celui des fonctionnalités d'un outil. La cellule d'état lit le tribunal, et vous pouvez demander au monde ce qu'il propose avant qu'on lui demande d'agir.
Le concepteur	Votre interface a une constitution de vingt-deux verbes et une grammaire fermée qui ajoute des rendus et jamais des verbes. Chaque élément doit dire quel verbe il sert, et la bibliothèque trouve déjà ceux qui n'en disent aucun.
Le décideur	Demandez de tout environnement trois choses. Sa barre d'état lit-elle un tribunal ? Son assistant vit-il dans un volet et propose-t-il ? Et qui juge que les deux shells s'accordent ? Ici les réponses sont : conçu ; conçu, avec la loi du rail construite en dessous au chapitre 20 ; et une page qui se vérifie elle-même, qui a attrapé le cas de la boîte vide quand je l'ai lue à la main le 8 octobre 2026 ; aucune garde ne la juge encore.
Le formateur	L'exercice est une prédiction : dites ce que le tribunal répondra avant de l'exécuter. Le mot d'état qui est une constante est une leçon sur la différence entre une étiquette et une vérité.
L'agent	Le shell est un fichier texte que vous pouvez lire et écrire, et l'arbre que lit un lecteur d'écran est l'arbre que vous pouvez lire. Vos propositions apparaissent dans un rail et ne valident nulle part.

9. EXERCICE

Un exercice, en trois mouvements.

1. Écrivez un shell à vous en cinq éléments : une barre de titre, un ruban de trois facettes tirées de la vie de votre propre solution, une zone de travail, un rail, une barre d'état. Avant d'exécuter quoi que ce soit, écrivez sur papier quel verbe parmi les vingt-deux chaque élément sert, et laquelle de vos boîtes vous vous attendez à ce que le tribunal appelle orpheline.
2. Exécutez `IsClean()` et `Orphans()`, et comparez avec votre papier. Puis cassez le fichier de trois façons : un enfant déclaré nulle part, un rôle qui n'est pas dans la liste, et une intention qui n'est pas l'une des vingt-deux. Pour chacune, écrivez ce que votre fonction d'état répondra, et si la bibliothèque, la fonction du livre, ou personne ne l'attrapera.
3. Émettez la page, ouvrez-la, et écrivez la ligne que la bande de la fixture imprimera pour une version avec une zone de travail vide. Puis regardez. La page vient de `ToFixture`, qui prend la boîte que `BoxOf` rapporte pour chaque nom sur le panneau mis en page par `ToPanel` ; `gui_profile_narrated` a la boucle, et un fichier de police est nécessaire.

HARO

```
oMine = new stzUiDocument('DEFINE PANEL shell ( SIZE [640, 360], DIRECTION column, CHILDREN [bar]
) RATIONALE "Mine."' + nl + 'DEFINE TEXT bar ( CONTENT "mine", SATISFIES understand ) RATIONALE
"Mine too."')
# under each line, write what you expect before you run it
? oMine.IsClean()
? @@( oMine.Orphans() )
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Si une promesse casse, ne la changez pas : expliquez d'abord la sortie.

Je vous laisse celui-ci. Le `ShellStatus` de la section 3 est la fonction du livre, et je ne l'ai pas gardée contre le cas ci-dessous : celui que la section 3 a passé par `IsClean()`, et que la section 7 range parmi les absents. Exécutez la ligne et lisez le mot que la cellule donne. Puis faites refuser par la cellule un étranger aux vingt-

deux, comme le fait la boucle du livre, avec la boucle sur la liste que la section 3 contient déjà, et sans demander à la bibliothèque une liste qu'elle ne possède pas (la section 4 dit pourquoi).

HARO

```
# the book's own call, with the one change of the second block of section 3
? ShellStatus(StzReplace(cShell, "SATISFIES preview", "SATISFIES transfer_funds"))
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Quand la cellule sera juste, vous saurez ce que la bibliothèque trouve, et ce que seule une liste qu'elle ne tient pas peut refuser.

Chaque shell de ce chapitre est une projection, et le chapitre 27 descend sous elles toutes : trois couches, un moteur, Haro. Là, chaque fonctionnalité est implémentée une fois, la surface est la promesse, et le runtime est la partie qu'on mesure, et qu'on n'affirme jamais.

> **OÙ SONT LES PREUVES**
>
> | | |
> |---|---|
> | NARRATION | `stzApp-building-your-first-app-narration.md` |
> | NARRATION | `stz-emulating-the-whole-solution-narration.md` |
> | NARRATION | `stz-system-dev-to-deploy-narration.md` |
> | SUR LE SITE | zui |
> | DÉCISIONS DE L'ÉTUDE | 12.12, 12.23, 12.24, 12.25 |

PARTIE V · CHAPITRE 27

Le moteur sous-jacent : trois couches, un moteur, Haro

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 9 bloc(s) sur 9 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je charge les trois couches de la bibliothèque, un fichier d'entrée à la fois, et je compte ce que chacune contient. Je lis le moteur en dessous, ses quatre-vingt-neuf bibliothèques et les fonctions qu'il enregistre, et je laisse le tribunal dire s'il est optimisé. Je pose les décisions, que l'avenir est Haro et que chaque fonctionnalité est implémentée une fois, dans le moteur, et j'imprime les lignes où un pair rapide est encore en tête. Le principe tient au-dessus de tout cela : la surface est la promesse, et le runtime est la partie qu'on mesure, et qu'on n'affirme jamais.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Une plateforme dans une seule base de code se dessine comme une pile avec une seule règle : une couche interne ne sait rien d'une couche externe. Le pair le plus net, tel que mon étude du dossier le dit, est celui des trois crates de Rust pour trois profondeurs d'hôte, choisies par la cible ; Rust impose la règle par son compilateur, ce qui est connaissance générale du terrain. Softanza le tient par des fichiers d'entrée, et la section 7 dit ce que cela coûte.

L'habitude du terrain pour un moteur est une bibliothèque native avec une interface C simple, enveloppée finement par chaque langage. SQLite est l'exemple courant, bien établi sur le terrain : une bibliothèque C, des liaisons dans presque tous les langages. Le dossier énonce la même conception pour son moteur : les clients de langage sont de fins ponts qui ajoutent syntaxe et paradigmes, et jamais de calcul.

Et le terrain a fait, à plusieurs reprises, une troisième chose : un langage garde sa surface et ses bibliothèques et change la machine en dessous. L'exemple propre du dossier est un descendant typé de Lua. Un autre, bien établi sur le terrain, est l'implémentation alternative de Python. Les deux enseignent la même chose que le dossier enseigne : la surface est la promesse, et le runtime est la partie qu'il faut mesurer.

Le principe qui en découle tient en une phrase, et la section 2 la pose.

2. LE PREMIER PRINCIPE

Chaque fonctionnalité est implémentée une fois, dans le moteur. La surface est la promesse, et le runtime est la partie qu'on mesure, et qu'on n'affirme jamais.

La première phrase est la règle de conception de l'architecture du moteur, dans un document de session : chaque fonctionnalité est implémentée une fois, dans le moteur. Les clients de langage sont de fins ponts qui ajoutent syntaxe et paradigmes, jamais de calcul. Le chapitre 4 a raconté ma décision qui en découle : j'ai refusé le calcul dans les faces. La seconde phrase est le cahier des charges du runtime en dessous : performance et portabilité, mesurées, jamais affirmées.

Et la définition qui gouverne les dépendances du moteur, que le chapitre 4 énonce déjà : la souveraineté est de n'avoir rien qu'un autre puisse retirer. Dans les mots de la vision, elle n'a jamais voulu dire que chaque ligne est de nous.

3. LA FORME SOFTANZA

Trois affirmations, et un bloc pour chacune de ce qu'un bloc peut prouver. Je vous dis d'abord ce qu'un bloc ne peut pas prouver, parce que cela compte. La règle de dépendance des couches, qu'aucun fichier noyau ne charge base et qu'aucun fichier base ne charge max, demande un balayage d'environ huit cent soixante-dix lignes de chargement. Seule une garde pourrait le porter, et il n'en existe aucune. J'ai fait le balayage avec un script à moi, et la section 7 en donne le résultat.

UNE COUCHE EST LE FICHIER QU'ON CHARGE

Combien de classes chaque fichier d'entrée charge-t-il ? Trois fichiers d'entrée, trois processus neufs. Chacun imprime combien de classes il contient, et si `stkList`, `stzList` et `stzListXT` existent. Le bloc lance des processus enfants, parce que le processus de ce chapitre a déjà chargé la plus grande couche, et que seul un enfant peut montrer le noyau seul :

HARO

```
cAsk = 'aC = ringvm_classesList()' + nl +
       '? "" + len(aC) + " " + (find(aC, "stklist", 1) > 0) + " " + (find(aC, "stzlist", 1) > 0)
+ " " + (find(aC, "stzlistxt", 1) > 0)' + nl
aEntries = [ "core/stkLib.ring", "stzLib.ring", "max/stxLib.ring" ]
for i = 1 to 3
    write("layer" + i + ".ring", 'load "../../../' + aEntries[i] + '"' + nl + cAsk)
    ? trim(systemcmd("ring layer" + i + ".ring"))
    remove("layer" + i + ".ring")
next
#--> 17 1 0 0
#--> 674 1 1 0
#--> 714 1 1 1
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le premier processus imprime `17 1 0 0`. Le noyau contient 17 classes, dix au préfixe `stk` et sept à `stzCore`, et aucune classe de base. Le deuxième imprime `674 1 1 0` : la base en contient 674 et aucune classe max. Le troisième imprime `714 1 1 1` : max en contient 714. Chaque couche inclut celle du dessous et aucune celle du dessus : noyau dans base dans max, chargées par le bas. Un chargement du seul fichier d'entrée du noyau prend environ un dixième de seconde sur cette machine. Un chargement de la base prend entre trois et huit secondes, en exécutions uniques le 9 octobre 2026. Mes notes consignent ces temps, et aucun bloc ne les montre.

Le contrat dit que la base s'appuie sur le noyau, par héritage ou par délégation. Que disent les classes ? `ringvm_classesList()` renvoie, pour chaque classe, son nom, son identifiant et son parent :

HARO

```
aClasses = ringvm_classesList()
? ParentOf("stkString")
#--> stzcorestring
? ParentOf("stzString")
#--> stzobject
? ParentOf("stzList")
#--> stzobject
? ParentOf("stzListXT")
#--> stzlist

func ParentOf(cClass)
    n = find(aClasses, lower(cClass), 1)
    if n = 0  return "(not loaded)" ok
    return aClasses[n][3]
```

✓ exécuté : a affiché ce que ce bloc promet

La deuxième ligne imprime `stzobject` : `stzString` hérite de `stzObject`, et non de `stkString` ! Le noyau porte une hiérarchie parallèle, dans laquelle `stkString` hérite de `stzCoreString`, le `stzcorestring` de la première ligne. Max s'appuie bien sur la base : `stzListXT` hérite de `stzList`, le `stzlist` de la dernière ligne. Exactement deux classes de la base héritent d'une classe du noyau, `stzSciNumber` et `stzPointer`, et une délègue, dans une ligne de `stzPointer`. Ce sont deux parmi les centaines de la base. Le contrat n'est pas rempli, et il ne l'est pas de la manière dont le chapitre 4 posait la question. Je l'imprime comme une dette.

Et la troisième couche par son préfixe. Dans le processus de la plus grande couche, combien de classes commencent par `stk`, combien par `stz`, et combien par `stx` ?

HARO

```
nStk = 0  nStz = 0  nStx = 0
aClasses = ringvm_classesList()
for aClass in aClasses
    cPrefix = left(aClass[1], 3)
    if cPrefix = "stk"  nStk++
    but cPrefix = "stz"  nStz++
    but cPrefix = "stx"  nStx++
    ok
next
? @@([ nStk, nStz, nStx ])
#--> [ 10, 693, 0 ]
```

✓ exécuté : a affiché ce que ce bloc promet

Le bloc imprime `[ 10, 693, 0 ]`. Dix, six cent quatre-vingt-treize, et aucune ! Les trois ne font pas les 714 du premier bloc : les classes qui restent ne portent aucun des trois préfixes, et un décompte par préfixe n'en dit rien, ce qui est le point de l'exercice. Les quarante classes que max ajoute à la base portent toutes le préfixe de la base. Les trente-deux classes au troisième préfixe existent, toutes dans un seul répertoire, et aucun fichier d'entrée ne les charge. La section 7 le redit.

UN MOTEUR

Les couches sont réelles au niveau du binaire. Le fichier de construction déclare les domaines du noyau et les domaines de la base et construit une bibliothèque par domaine, dans une boucle. Combien de bibliothèques déclare-t-il pour chaque couche ? Ce bloc lit ce fichier, et l'affirmation porte sur ce fichier :

HARO

```
$cEngineDir = "../../../engine"
cBuild = read($cEngineDir + "/build.zig")
n1 = substr(cBuild, "const core_domains")
n2 = substr(cBuild, "const base_domains")
n3 = substr(cBuild, "fn addSpeech")
? Q(substr(cBuild, n1, n2 - n1)).NumberOfOccurrence('.name = "stk_')
#--> 4
? Q(substr(cBuild, n2, n3 - n2)).NumberOfOccurrence('.name = "stz_')
#--> 85
? Q(cBuild).NumberOfOccurrence('stx_')
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche files ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le bloc imprime `4`, `85` et `0`. Quatre bibliothèques noyau, quatre-vingt-cinq bibliothèques base, et aucune bibliothèque pour la troisième couche : quatre-vingt-neuf en tout. Le repère `fn addSpeech` est la première fonction après les deux tables,

une frontière en texte brut, arbitraire mais visible. Un tribunal des couches ferait une analyse correcte, et je le range parmi les demandes. Mon étude du dossier dit quatre-vingt-six domaines de base à ce commit, et j'en lis quatre-vingt-cinq.

Le moteur dit à la bibliothèque ce qu'il est. Sa version est une constante. Combien de fonctions distinctes enregistre-t-il dans la machine virtuelle une fois chargé ?

HARO

```
? StzEngineEmbedVersionString()
#--> 0.8.0
aAll = ringvm_cfunctionslist()
aNames = []
for cName in aAll
    if left(lower(cName), 9) = "stzengine"  aNames + lower(cName)  ok
next
? len( Q(aNames).DuplicatesRemoved() )
#--> 2786
```

✓ exécuté : a affiché ce que ce bloc promet

Le bloc imprime `0.8.0` et `2786`. Deux mille sept cent quatre-vingt-six fonctions distinctes du moteur, la vue propre de la bibliothèque, prise à l'exécution. Le même appel renvoie plus d'entrées, puisque celles de la machine virtuelle elle-même sont incluses, et plusieurs noms apparaissent plus d'une fois ; je n'ai pas cherché pourquoi. J'ai d'abord assigné la liste à une variable. L'appel est réévalué à chaque tour d'une boucle, et j'ai arrêté deux exécutions qui avaient grossi jusqu'à des gigaoctets avant que je l'apprenne. Le décompte bouge avec le moteur construit.

Il y a trois décomptes de trois choses différentes, et le chapitre dit lequel il imprime. Le premier est le décompte vu de la bibliothèque à l'exécution : les 2 786 ci-dessus. Le deuxième est le décompte vu de n'importe quel hôte par l'interface C. Un script à moi a lu les tables d'export des quatre-vingt-neuf bibliothèques le 9 octobre 2026 et y a trouvé 1 710 symboles nommés, dont quatre-vingt-neuf sont le crochet du chargeur, donc 1 621 fonctions. Le troisième est le décompte dans les sources Zig, comme noms d'enregistrement : 2 703 littéraux distincts. Le site dit 752 et 2 652, au commit 010743cce tel que la page le lit au 9 octobre 2026, et les constantes typées de

la bibliothèque disent 1 513, datées du 16 juin 2026. Je ne peux reproduire 752 par aucune méthode que j'ai essayée. N'imprimez qu'un décompte que la construction lit, avec sa méthode et sa date.

Le moteur fait le travail. Que fait le `upper` de l'hôte de l'eszett allemand, et qu'en fait la mise en correspondance de casse du moteur, sa propre table au-dessus d'une bibliothèque embarquée ? Une seconde bibliothèque embarquée extrait les radicaux des mots. Ce sont deux des dix-neuf bibliothèques embarquées du moteur, que la section 4 compte :

HARO

```
? upper("Straße")
#--> STRAßE
? Q("Straße").Uppercased()
#--> STRASSE
? @@( Q("running runs ran").StemmedWords() )
#--> [ "run", "run", "ran" ]
```

✓ exécuté : a affiché ce que ce bloc promet

La première ligne imprime `STRAßE` : l'hôte laisse l'eszett tranquille. La deuxième imprime `STRASSE` : le moteur écrit SS ! La troisième imprime `[ "run", "run", "ran" ]` : l'extracteur de radicaux ramène `running` et `runs` à `run` et laisse `ran`. Si le `upper` de Haro devient du Unicode complet, la promesse change.

Et le moteur est optimisé. Le mode par défaut de la construction est optimisé et sûr, depuis le commit 1fdc2eda5, que le journal git de la bibliothèque date du 26 juillet 2026. Les fichiers épinglés portent le commit et non la date. Jusque-là, chaque bibliothèque que la bibliothèque avait jamais chargée était compilée sans aucune optimisation, et cela clôt la question que le chapitre 22 avait laissée ouverte sur le mode de construction. Une garde lit le temps, puisqu'elle ne peut pas lire un drapeau. Vingt sommes sur deux millions de doubles qui vivent dans le moteur finissent-elles en moins d'un dixième de seconde, et sont-elles justes ?

HARO

```
oBuf = StzNumBufferOfSizeQ(2000000)
oBuf.RangeFrom(1, 1)
```

```
nT0 = clock()
for k = 1 to 20
    nSum = oBuf.Sum()
next
? (clock() - nT0) / clockspersecond() < 0.10
#--> 1
? nSum
#--> 2000001000000
oBuf.Free()
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the clock ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

Le bloc imprime `1` et `2000001000000`, le booléen et la somme, et aucun temps, parce qu'un temps n'est pas une promesse. Sur trois exécutions les 8 et 9 octobre 2026, les temps écoulés ont été de 0,02, 0,04 et 0,03 seconde, tirés de mes notes et absents du relevé d'exécution. La table propre de la garde dit 0,14 pour le mode non optimisé et 0,04 pour l'optimisé. La garde elle-même a tourné le 8 octobre : sept assertions dans le fichier, six atteintes ici, six réussies. La septième est sur une branche else qu'on n'atteint pas quand l'analyseur est construit.

CE QUI L'EXÉCUTE

Qu'est-ce qui exécute chaque bloc de ce livre ? Je demande sa version au runtime :

HARO

```
? version()
#--> 1.27
```

✓ exécuté : a affiché ce que ce bloc promet

Il imprime `1.27` : la version de l'ancien langage hôte, l'oracle contre lequel le nouveau runtime est jugé. Haro n'exécute pas la bibliothèque. Sous un futur runtime la promesse change, comme la ligne de preuve le dit.

Les décisions viennent ensuite, et la première porte sur cet avenir.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai décidé que l'avenir est Haro, et non l'ancien langage hôte. Mes mots du 7 octobre 2026, tels que mon étude du dossier les cite, les crochets de moi : l'avenir est Haro, pas [l'ancien langage], et le site doit être clair là-dessus. La face de Haro pour l'ancien langage ne sera maintenue que pour que le code déjà écrit continue de fonctionner. Ce qui n'est pas porté, c'est le runtime de ce langage et sa gouvernance. La première décision, le 28 août, était qu'il n'est pas porté du tout ; il a été remplacé deux jours après, comme l'amendement de la vision le dit.

J'ai décidé que la réécriture est un renommage. Le code de mes articles, les narrations, réécrit au nom de Haro, est le même que la face de l'ancien langage. La charte le mesure de même : les étapes propres de Haro exécutent le même texte, identique à l'octet sur le nouveau compilateur et sur l'oracle.

J'ai décidé que Haro est le langage des langages. L'en-tête de la charte cite ma note du 1er octobre 2026. La charte a été ratifiée ce jour-là, avec mon « go with your recommendations » sur sept décisions. Ses amendements du 4 octobre répondent à dix-neuf questions ouvertes du dossier, et la version reste ratifiée jusqu'à ce que j'aie lu sa section 15.

J'ai décidé que la surface est conservée. Sa surface aujourd'hui est celle que le compilateur compile. La bibliothèque est sa spécification de compatibilité. Les autres surfaces sont des faces sur un même graphe de programme : chaque syntaxe est une face, pour qu'un agent d'IA à l'aise avec l'ancien langage, avec Python ou avec JavaScript écrive du Haro. Parmi elles seule la première face existe. Une face en forme de Python est une orthographe, pas du Python.

J'ai retiré l'ancien cadre, sans repli. Moteur seul : pas de repli, pas de drapeaux de fonctionnalité, pas de dépendance à Qt. Le statut du 12 juin 2026 dit qu'il n'en reste aucun. J'ai vérifié. Aucun fichier chargé ne charge une bibliothèque

Qt. Deux extensions externes sont chargées par leur nom : une extension zip, que la base charge, et une de cryptographie, dans un répertoire qu'aucun fichier d'entrée ne charge.

J'ai ordonné le retrait des extensions propres du langage. Ma directive du 13 juin 2026 : s'assurer qu'aucune extension externe n'est requise, et que des alternatives du moteur les remplacent. La bibliothèque en charge encore une, une extension zip, comme l'entrée ci-dessus le dit.

J'ai mis le moteur en Zig, sans dépendance à une bibliothèque, comme mon étude du dossier rapporte ma demande. C'est qualifié par la définition de la souveraineté du dossier lui-même : dix-neuf bibliothèques embarquées, chacune avec un enregistrement.

J'ai laissé tout aller dans base. Ma phrase, comme mon étude du dossier la rapporte : sous la pression du temps, tout est allé dans base, en attendant le bon moment pour refactoriser à nouveau. D'après mon décompte plus bas, base représente quatre-vingt-treize pour cent des lignes.

J'ai choisi l'optimisation sûre. Ni le mode le plus rapide, ni le non optimisé. Je l'ai choisie parce qu'une panique propre vaut mieux qu'une corruption silencieuse de la mémoire.

J'ai refusé une affirmation phare qui ne se reproduit pas. Le mémo du 13 septembre 2026 dit que l'affirmation à retirer est que le runtime bat le pair rapide, au présent, sur x64. Le chapitre 22 a raconté le refus, et la section 6 imprime les lignes.

J'ai refusé un pourcentage. Le chiffre est de moi, et il ne devrait pas être imprimé sans la garde qui le produit. Les chapitres 31 et 33 le disent.

Je n'ai aucune phrase sur la façon dont une couche est choisie en Haro. Mon étude du dossier propose seulement qu'en Haro, le choix de la couche soit une déclaration que le compilateur honore. La charte n'a aucun texte sur noyau, base et max. La base héritant du noyau est le contrat tel que conçu, avec le mot d'étape conçu. Le code ne le respecte pas aujourd'hui, comme l'imprime la section 3, et le compte rendu du livre est ce que fait le code : deux implémentations sur un seul

moteur. Et en Haro, une couche est choisie aujourd'hui par le fichier qu'on charge. Une déclaration que le compilateur honore est la proposition de mon étude, et elle est conçue.

La section 5 lit ce que le tribunal prouve de tout cela, et ce qu'aucune garde ne prouve.

5. LE TRIBUNAL

`engine_is_optimised_narrated` prouve que le moteur est compilé avec optimisation : vingt sommes sur deux millions de doubles en moins d'un dixième de seconde, et la bonne réponse. Elle prouve qu'un indice de 99 et un indice de 0 sont refusés et le tampon intact, et que l'analyseur embarqué que le mode optimisé avait cassé répond maintenant. Je l'ai exécutée : six des sept assertions du fichier ont été atteintes, et six ont réussi. `engine_senses_narrated` prouve que le moteur donne au processus cinq sens : la mémoire, son pic, la mémoire du système, le temps du processeur, et une série bornée. Je l'ai exécutée : trente-six sur trente-six, et l'exécution s'arrête ensuite avec "STOPPED!" par conception. `wasm_narrated` prouve que l'entrée du moteur pour le navigateur exporte son interface choisie et que la construction a une cible pour elle. Elle construit un sous-ensemble et affirme qu'il est plus petit. Je ne l'ai pas exécutée, parce qu'elle appelle une construction de la chaîne d'outils. `threat_model_narrated` lit le modèle de menaces et vérifie que chaque ligne de garantie nomme une preuve et que chaque fichier de garde nommé existe. Les lignes G11, les harnais de fuzzing, et G25, la nomenclature de composants avec ses empreintes, concernent le moteur.

Ce qui n'existe pas : **aucune garde ne juge les couches.** Aucun fichier sous les répertoires de test ne charge le fichier d'entrée d'une couche ni ne lit le contrat. Il y a 529 gardes narrées dans base, et aucune dans noyau ou dans max. Mon étude du dossier dit 517 à un endroit et 387 à un autre, et le site 501 ; je n'imprime que ce que la construction lit. Et aucune garde ne juge les décomptes du moteur : les nombres qu'imprime la commande d'état en ligne de commande du moteur sont des constantes typées datées du 16 juin 2026.

La section 6 place le runtime à côté du terrain, et le pair est en tête sur chaque ligne.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`engine_is_optimised_narrated` 7 assertions
base/test/number/engine_is_optimised_narrated.ring

`engine_senses_narrated` 36 assertions
base/test/perf/engine_senses_narrated.ring

`wasm_narrated` 23 assertions base/test/system/wasm_narrated.ring

`threat_model_narrated` 11 assertions
base/test/security/threat_model_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

La vitesse. Le runtime en dessous contre un runtime de script rapide du terrain, le 13 septembre 2026. La méthode est celle du mémo : six paires de noyaux de calcul écrits à neuf, même algorithme et mêmes tailles des deux côtés, chaque côté dans son idiome rapide, sommes de contrôle entre langages vérifiées avant tout nombre. Les minima sont pris sur trois répétitions internes et deux exécutions de processus, le nouveau compilateur construit pour la vitesse, le pair en interpréteur seul sans JIT, sous Windows, x64. Le pair est en tête sur chaque ligne :

LIGNE	LE RUNTIME EN DESSOUS (MIN MS)	LE PAIR (MIN MS)	LE PAIR EST EN TÊTE DE
sieve	128	74,5	1,72
matmul	349	185,6	1,88
fib	57	50,2	1,14
mergesort	376	206,7	1,82

LIGNE	LE RUNTIME EN DESSOUS (MIN MS)	LE PAIR (MIN MS)	LE PAIR EST EN TÊTE DE
binsearch	536	513,1	1,04
bytescan	53	37,5	1,41

Les mots du mémo : le pair est en tête sur chaque ligne, fib et binsearch sont des quasi-égalités, et rien n'est pire que 2x. ARM n'est pas mesuré. La victoire que le mémo garde : à moins de 2x sur chaque ligne, ce qu'aucune version de l'ancien runtime n'a jamais été.

L'affirmation phare est encore vivante dans les notes propres du dépôt. Comme je l'ai lu le 9 octobre 2026, le cahier des charges du dépôt propre du runtime et ses notes disent encore que le spike bat les pairs sur chaque ligne. La correction est portée dans un autre dépôt du domaine de projets et dans le mémo. Dans le dossier propre du runtime, elle n'est portée que pour sa configuration de version candidate, dans une section des notes. Le mémo l'a demandée. Le mémo du 13 septembre 2026 retire l'affirmation, et les six lignes ci-dessus sont les siennes, bien que les notes du dépôt portent encore l'ancienne affirmation. Je ne nomme aucun pair, et je le décris comme le fait le chapitre 22 : un runtime de script rapide du terrain.

La mémoire. Le même jour la note dit : résidente, exécutant la bibliothèque, ce matin 733 mégaoctets, maintenant 149, et l'ancien runtime 139. Une perte de dix contre l'ancien runtime, après une chute de 584. Une victoire et une perte dans une même note.

La compatibilité, le 5 octobre 2026. La suite large de la bibliothèque a été exécutée sur un instantané. La note consigne la parité des assertions sur chacune de celles que l'ancien runtime passe, que la plupart des programmes s'accordent sur chaque ligne stable, et que dix-neuf lignes diffèrent par nature. La note ne dit pas Windows dans cette entrée, et la passe Linux est listée comme prochaine étape. Ce sont des notes de session d'une branche privée, et aucun nombre de moi. Je les imprime, datés du 5 octobre 2026, avec le mot d'étape en développement : 8 540 assertions sur 8 540 à parité, 1 253 tests sur 1 253 qui portent des assertions, et 3 425 programmes sur 5 112 identiques sur chaque ligne stable. Il y en a 1 267 de plus, identiques sauf sur la ligne qui imprime le temps écoulé, et 19 lignes diffèrent par

nature : 13 impriment davantage, 5 sont refusées ou lèvent une erreur, une écrit une ligne sur la sortie d'erreur. Ce sont les décomptes de la note ; la note ne place pas les programmes restants, et je n'en ai exécuté aucun.

La plateforme. Construite et exécutée sous Windows aujourd'hui. Une construction Linux de quatre-vingt-cinq des quatre-vingt-neuf modules existe sur une branche de travail ; le site dit qu'elle est en construction. Un programme entier avec son moteur dans un seul fichier, sans dépendance, est absent.

Le moteur contre le terrain. La ligne du modèle de menaces pour le fuzzing le dit elle-même : pas de fuzzing planifié ni guidé par la couverture, pas d'intégration continue et des commits non signés, que j'ai différés pour la phase de développement. Les bibliothèques natives maintenues du terrain ont du fuzzing continu et des versions signées. Et la perte du chapitre 22, inchangée : pas de collation par locale, et pas de fuseaux horaires.

La section 7 dit où s'arrêtent les couches et le moteur.

7. LA LIMITE

CONSTRUIT — trois couches choisies par le fichier d'entrée qu'on charge, chacune incluant celle du dessous et aucune celle du dessus ; la règle de dépendance tenue aux chargements ; un moteur de quatre-vingt-neuf bibliothèques, en Zig, construit en mode optimisé sûr, avec le propre décompte du moteur de ce qu'il expose ; Haro comme charte ratifiée et un spike dont les tribunaux tournent sur le nouveau runtime et sur l'ancien

CONÇU — la couche base héritant de la couche noyau ; la couche max comme couche construite ; un tribunal qui vérifie les couches contre leur contrat ; une couche choisie par une déclaration que le compilateur honore ; les autres faces de Haro sur un même graphe de programme ; un chargeur pour un domaine de la base ; le dépôt autonome du moteur et sa ligne de commande ; le pont polyglotte du moteur, qui est celui du chapitre 16

ABSENT — une couche noyau légère qui exécute ses chaînes ; un moteur Linux ou macOS en service ; une passe mesurée de la bibliothèque sur le nouveau runtime sous Linux ; un exécutable `haro` ; Haro exécutant la bibliothèque ;

un exécutable unique sans dépendance ; un décompte de tests du moteur que je puisse imprimer

La couche légère ne tourne pas. Que rend une chaîne du noyau quand je lui demande son contenu ?

HARO

```
o = new stkString("abc")
? o.Content()
#--> Error (R3) : Calling Function without definition: stkenginestringfrom
```

✓ exécuté : a affiché ce que ce bloc promet

Le runtime répond : `Error (R3) : Calling Function without definition: stkenginestringfrom`. La chaîne du noyau ne peut pas rendre `abc` ! La classe chaîne du noyau appelle des noms de moteur qu'aucune bibliothèque du moteur n'enregistre. Dans un processus du noyau seul, la bibliothèque noyau du moteur enregistre cinq cent trente-quatre fonctions sous les noms de la base et aucune sous les noms du noyau. Ce décompte vient de mes notes, et aucun bloc ne le montre. Donc `stkString` et `stkChar` ne trouvent rien. Mon étude du dossier le range comme point ouvert du noyau et du moteur. `stkList` tourne, ce que le chapitre 4 a montré, et le chargement de dix-sept classes en un dixième de seconde est réel.

La couche légère n'est pas légère au niveau du binaire, pour les chaînes. Par taille de fichier, sur la construction du 5 octobre 2026, la bibliothèque chaîne du noyau fait 22 355 968 octets et celle de la base 22 362 624. Les données en lecture seule font 20,8 mégaoctets dans les deux ; je n'ai pas diagnostiqué ce qu'elles contiennent. Les quatre bibliothèques noyau totalisent 24,1 mégaoctets sur les 122,4 des quatre-vingt-neuf bibliothèques. Les bibliothèques noyau pour les dates, les fichiers et la locale diffèrent de leurs jumeaux de la base de moins d'un pour cent.

La règle de dépendance tient aux chargements, avec une exception à un appel. J'ai analysé chaque chargement de chaque fichier non test, non archive du noyau, de la base et de max, 871 lignes, et résolu chacun. Le noyau ne charge, hors de son propre répertoire, que la face du moteur ; la base charge le noyau et le moteur ; aucun fichier sous la base ne charge quoi que ce soit sous max ou future ; max

charge la base. Mais `FindMin`, dans la liste de nombres du noyau, appelle une fonction qui n'est définie que dans la base. Dans un processus du noyau seul, `FindMin([4,2,9])` répond 2, et `FindMin("not a list")` lève "Calling Function without definition". Dans un processus de base il lève l'erreur voulue. J'ai trouvé une instance par une recherche de noms de la base dans le noyau, et je n'ai pas cherché d'autres.

Sur un disque sensible à la casse, le fichier d'entrée du noyau ne charge pas. Sa deuxième ligne charge un fichier par un nom dans la mauvaise casse. Mon balayage trouve vingt tels chargements dans des fichiers non test à ce commit ; le dossier propre du runtime en a compté cinquante-cinq avec les tests, le 8 octobre 2026. Le livre est construit sous Windows, qui le cache.

La troisième couche est écrite et non câblée. Ses trente-deux classes sont toutes dans un seul répertoire, et aucun fichier d'entrée ne les charge. La première d'entre elles fonctionne chargée à la main. Je l'ai exécuté et je ne le montre pas, une chaîne de onze caractères comptée par elle. Je n'ai essayé aucune des trente et une autres. Une couche de bibliothèque pour elles dans le moteur n'existe pas.

Ce qui est construit : une couche est un répertoire, un préfixe et un fichier d'entrée. J'ai compté le 9 octobre 2026 à ce commit, par fichiers avec l'extension, en excluant les répertoires de test et d'archive. Noyau, 27 fichiers et 6 366 lignes ; base, 984 fichiers et 519 579 ; max, 115 et 31 292 ; et un répertoire de l'avenir, 51 fichiers et 6 356. La base représente 93,2 pour cent des lignes des trois couches. Le moteur : quatre-vingt-neuf bibliothèques issues des sources Zig, 401 fichiers source, 179 985 lignes, et 2 464 tests déclarés, que je n'ai pas exécutés. La chaîne d'outils est Zig 0.15.2 d'après le propre commentaire du fichier de construction. Dix-neuf bibliothèques embarquées, chacune avec un enregistrement de composant, version, source, licence et empreinte. Les lacunes propres du dossier : une empreinte d'archive est consignée pour neuf des dix-neuf, et trois n'ont pas de version.

Ce qui est conçu : la couche base héritant de la couche noyau. Les sept domaines partagés sont implémentés deux fois sur le même moteur. Max comme couche construite. Un chargeur pour un domaine de la base. Un tribunal qui vérifie les

couches contre leur contrat. Le dépôt autonome du moteur et sa ligne de commande, que l'architecture liste comme une phase. Le pont polyglotte du moteur, qui est celui du chapitre 16.

Ce qui est absent : un moteur Linux et macOS en service ; une passe mesurée de la bibliothèque sur le nouveau runtime sous Linux ; un exécutable unique sans dépendance ; un décompte de tests du moteur que je puisse imprimer.

HARO

Je dis ce qui est construit, ce qui est conçu, et ce qui est absent, tel que le dossier l'a au 9 octobre 2026.

Construit, comme une charte, une famille de grammaires et une garde. La charte, version 0.2, ratifiée le 1er octobre 2026. Un spike de dix-neuf étapes, que le dossier appelle des échelons, entre le 1er et le 4 octobre, chacun un document de grammaire avec des fixtures. Sa garde a rapporté un succès de l'exécution entière le 5 octobre. Ses tribunaux sont des programmes qui tournent à l'identique à l'octet sur le nouveau compilateur et sur l'ancien runtime officiel. Le nouveau compilateur met aussi en forme et exporte les fichiers de la bibliothèque : sur environ six mille, presque tous impriment les octets de la référence, et quelques dizaines sont refusés par les deux. Une première tranche d'un graphe du code a été construite le 5 octobre.

Ce qu'est Haro aujourd'hui, en une phrase que le dossier soutient : une charte ratifiée et un spike dont les tribunaux tournent sur le nouveau runtime et sur l'ancien, sur la surface dans laquelle la bibliothèque est écrite. Il n'y a pas d'exécutable `haro`. Le répertoire des binaires contient le compilateur et deux autres. Les tribunaux forgés ne chargent aucune bibliothèque. **Haro n'exécute pas la bibliothèque.** La vision du 27 septembre disait que Haro n'existe pas et que la ligne Linux du nouveau runtime n'est pas mesurée. La première moitié est périmée, puisqu'une charte et un spike existent. La seconde moitié reste vraie de la suite large sous Linux.

Conçu : les trois altitudes, déclarant, déclaré, et calculant ; les six lettres de son alphabet, que la charte nomme, cinq d'entre elles forgées entières dans le spike ; le raffinement comme surface d'écriture ; d'autres faces sur un même graphe de

programme. Ces faces arrivent comme imprimeurs avant d'arriver comme analyseurs, et jamais comme sémantique. Les verbes `haro forge`, `check`, `refine`, `run`, `project` et `narrate` sont une recommandation, et non un programme.

Le runtime en dessous, tel que le dossier le date. Son cahier des charges : une machine à registres et un compilateur du source de l'ancien langage vers son bytecode, exécutant les programmes existants. Environ trois cent mille lignes de Softanza sont sa spécification de compatibilité. Sa face publique : la branche principale est publique et intacte ; le travail vit sur une branche poussée vers un dépôt distant privé seulement. Son dernier commit à ma lecture date du 8 octobre 2026, à 22 h 26. Le mot d'étape est de moi, et c'est « en développement ».

Les six portes suivent, une pour chaque lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous choisissez votre profondeur par le fichier que vous chargez, et vous pouvez le prouver en trois processus neufs. La couche légère n'exécute pas ses chaînes aujourd'hui : prenez la base. Lisez le décompte du moteur comme la bibliothèque le voit, et non comme une page le tape.
L'analyste	Trois nombres pour « quelle est la taille du moteur », et trois méthodes. Demandez laquelle un fournisseur veut dire. La bibliothèque noyau a la taille de celle de la base pour les chaînes.
Le concepteur	Une couche est une décision sur ce que vous livrez, et non sur le goût. Au niveau du binaire, le noyau et la base ont une bibliothèque par domaine et la troisième couche n'en a aucune ; le sous-ensemble web, que les chapitres 4 et 24 montrent, est un sous-ensemble et non une couche.
Le décideur	Les deux questions du chapitre 4, plus deux. Ce qui l'exécute : la version 1.27 aujourd'hui, avec le remplaçant en développement. Ce que le remplaçant mesure : à moins de 2x d'un pair rapide sur chaque ligne, de 1,04 à 1,88 fois, ce qu'aucune version de l'ancien runtime n'a jamais été, et dix mégaoctets de mémoire de plus que l'ancien. C'est mon point de vue : un dossier qui imprime sa perte est le seul genre à croire.
Le formateur	Enseignez la règle de dépendance en faisant charger le noyau seul à l'apprenant et prédire, puis compter. Enseignez « mesuré, jamais affirmé » avec le mémo du 13 septembre 2026.
L'agent	Vous pouvez compter vous-même les fonctions du moteur. Ne répétez pas un décompte d'une page, et n'écrivez pas une boucle directement sur la liste des classes. Dites au lecteur dans quelle couche vit une classe par sa chaîne de parents, et non par son préfixe.

9. EXERCICE

Avant d'exécuter quoi que ce soit, écrivez trois nombres : combien de classes chaque fichier d'entrée chargera, et lesquelles de `stkList`, `stzList` et `stzListXT` chacun contient.

HARO

```
# core/stkLib.ring, stzLib.ring, max/stxLib.ring: write your three predictions here,
# then load each entry in its own process and count with ringvm_classeslist().
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Puis choisissez n'importe quelle classe de la bibliothèque, prédisez son parent, et lisez-le avec la chaîne de parents. Écrivez la ligne que votre prédiction a ratée. Un apprenant qui devine que `stzString` hérite de `stkString` est en bonne compagnie : le contrat le dit, et la bibliothèque non. Je ne dis pas lequel des 17, 674 ou 714 est lequel.

Je vous laisse celui-ci. Le décompte par préfixe de la section 3, `[ 10, 693, 0 ]`, est juste, et il ne dit rien des couches. Écrivez ce décompte à nouveau pour qu'il interroge la chaîne de parents de chaque classe, lue avec `ParentOf`, et non ses trois premières lettres, et regardez ce que vous obtenez. La section 3 dit ce que le préfixe cache, et la section 8, la porte de l'agent, dit pourquoi je le refuse. Quand votre décompte répondra, vous saurez pourquoi la couche d'une classe se lit dans ce qu'elle hérite, et non dans le nom qu'elle porte.

Le chapitre suivant quitte le moteur pour un restaurant à Lyon, et pour la phrase que le projet y a enseignée : livrer petit, vite et utile, et laisser les fondations mûrir en parallèle sans jamais bloquer une livraison.

OÙ SONT LES PREUVES

NARRATION	`stz-engine-senses-narration.md`
NARRATION	`stz-vs-qt-unicode-narration.md`
NARRATION	`stz-emulating-the-whole-solution-narration.md`
NARRATION	`stz-honest-stopwatch-narration.md`
SUR LE SITE	architecture
DÉCISIONS DE L'ÉTUDE	12.10, 12.11, 12.15, 12.21, 12.28, 12.29

PARTIE VI

Des artisans du logiciel, forgés

28 · Un restaurant à Lyon
29 · Une banque à Niamey
30 · Une organisation sur le terrain
31 · Ce que les projets montrent, et la suite

PARTIE VI · CHAPITRE 28

Un restaurant à Lyon

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 5 bloc(s) sur 5 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je raconte une leçon payée dans l'ordre, avec ses dates : un restaurateur de Lyon, deux ans de spécification sans rien d'utilisable, et le renversement du 4 juillet 2026, le client en premier. Je montre ce que la bibliothèque en porte : le Verrou comme règle sur les mondes d'un cycle, un bac à sable de paiements non complaisant, une constellation qui endort un monde, un budget de vitesse déclaré et jugé. Je dis où elle ne porte rien, et les six choses que j'ai refusées, parmi lesquelles le grand système d'un seul bloc, la dérive silencieuse, le faux complaisant, et prêter au propriétaire ce qu'une simulation avait dit. Le principe tient en une phrase sur le site : livrer petit, vite et utile, et laisser les fondations mûrir en parallèle sans jamais bloquer une livraison.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Chaque leçon de ce projet a, sur le terrain, un lieu ordinaire, et le terrain la garde en dehors du programme.

Un périmètre fixé pour un cycle vit dans un processus. En Scrum, le but d'un sprint ne change pas pendant qu'il court ; en Shape Up, le temps est fixé et le périmètre varie, et les idées nouvelles attendent hors du cycle. Essayer un service avant de le payer vit dans le mode test d'un fournisseur et dans les outils qui tiennent la place d'un service HTTP. Un budget de performance vit dans un contrôle d'intégration continue du poids et de la vitesse d'une page. La mesure de la vitesse propre au web est un chiffre pris sur de vrais chargements de pages, au soixante-quinzième

percentile. L'image de toute une constellation de petites applications vit dans la super-application : un hôte qui embarque un runtime et fait tourner des mini-applications à l'intérieur. Tout cela est une connaissance générale, pas une étude du dossier.

Les quatre sont de bonnes pratiques, et dans les quatre le programme lui-même n'en sait rien.

2. LE PREMIER PRINCIPE

Le projet fut une leçon payée, et la leçon tient en une phrase sur le site : **livrer petit, vite et utile, et laisser les fondations mûrir en parallèle sans jamais bloquer une livraison.**

Sa compagne est la règle de preuve de ce chapitre, prise sur la même page : une leçon n'est listée que lorsqu'un fichier de la bibliothèque la porte.

3. LA FORME SOFTANZA

Un restaurateur de Lyon possède un restaurant de couscous, et il voulait une plateforme pour le commerce de quartier. Il n'est pas programmeur, et les notes du projet le disent. Je vais raconter le projet dans l'ordre, avec ses dates, puis montrer ce que la bibliothèque porte de chaque leçon.

Tout a commencé par une offre, le 2 mai 2024, pour une mission d'analyse métier. Ce qui a suivi, ce sont deux ans, de l'aveu même du propriétaire, de recherche et de réflexion, et de ce que j'appelais la spécification : des analyses fondatrices, un roman de narrations, un premier produit spécifié jusqu'au dernier détail. Cela n'a rien produit qu'il puisse utiliser. Au milieu de 2025, le projet a failli s'arrêter, et un silence de six mois a suivi. C'est là que j'ai appris la doctrine de la section 2 : livrer petit, vite et utile.

Le 3 juillet 2026, nous avons répété le projet à deux : un cadrage de ses réponses que j'ai écrit avec une session, et une campagne de vingt-deux tests simulés, menés par des sessions jouant le propriétaire. Je dis « simulés » chaque fois, parce que les réponses de ce fichier ne sont pas les siennes. Le lendemain, 4 juillet, il a répondu avec ses propres mots à un vrai rapport de cadrage. Il a renversé l'ordre du travail.

Nous avions imaginé une ligne qui allait du fournisseur au restaurateur et finissait par le client. Il l'a retournée : le client d'abord, le restaurateur ensuite, le fournisseur en dernier. Et il a écrit que la vitesse de commande était sa première priorité, non négociable. Sa grande crainte était la friction au premier chargement, et la seconde crainte, dans le même rapport, était le calendrier : aucune des échéances jusque-là n'avait été tenue.

La réponse du projet à sa crainte est un document et un carnet. Nous l'avons appelée le Verrou. En juillet, un second cadrage la lui a proposée, clause par clause. Les trois premières : un seul avenant ; toute idée nouvelle va dans un carnet, relu à la fin du cycle ; une entrée exige une sortie écrite. Les deux dernières : une démonstration toutes les deux semaines et aucune date au-delà du prochain cycle ; et une définition de « terminé ». Il a accepté quatre clauses telles qu'écrites et ajusté la cinquième en y ajoutant un critère de lui : que d'autres commerçants et partenaires puissent installer le produit seuls. Dans le dossier, c'est moi qui ai demandé le Verrou en premier, comme une règle sur moi-même : que le changement de cap soit le seul changement de cap permis. Le 4 août 2026, le premier avenant a été clos. Douze décisions ont été scellées ; trois ont été reposées en termes simples, parce que lui-même avait signalé qu'il ne les avait pas comprises. La leçon notée porte sur la manière de demander : dès qu'une question parle de jalons, de phasage et de cycles, le propriétaire cesse de suivre. Alors les questions sont passées dans la langue de son métier.

Le 15 août, la première démonstration a été conçue pour lui seul. La liste de contrôle porte son nom et « seul » dans son titre. Il ouvre la livraison sur son propre ordinateur, scanne un code avec son téléphone et un autre avec sa tablette, passe sa propre commande depuis le téléphone, et la regarde arriver sur la tablette. S'il reste bloqué, un bouton prépare un message à coller dans une messagerie. Le critère de réussite se lit : il commande depuis son téléphone, et cela arrive sur sa tablette, sans nous. Le carnet montre que cela s'est produit : ses remarques de ce jour y ont été saisies le 17. Et le 19 août, il a passé une journée de terrain avec deux téléphones et des photographies envoyées par une messagerie, à mesurer sur un vrai téléphone. Un point d'accès fonctionne sans carte dedans ; l'adresse de la passerelle était stable d'un redémarrage à l'autre. Un téléphone est client Wi-Fi ou point d'accès, jamais les

deux ; le système d'un iPhone ne revient pas de lui-même sur un réseau sans internet. Les notes du projet l'appellent alors son co-architecte. Le dossier s'arrête le 25 août 2026.

Voilà l'histoire. Qu'en porte la bibliothèque ?

Le Verrou. Je dois être exact ici. La page Forged du site contient une règle que j'ai faite : une leçon n'est listée que lorsqu'un fichier de la bibliothèque la porte. Aucun fichier ne porte le Verrou. C'était un pacte et un carnet, tenus dans des documents. Ce que la bibliothèque peut faire, c'est vous donner la déclaration qui s'en approche le plus : une règle sur les mondes d'un cycle, selon laquelle un monde sans admission écrite est un constat. Que dit le tribunal quand une idée entre dans le cycle sans admission écrite ? Ce bloc est une construction, pas le code du restaurant :

HARO

```
oRule = new stzGraphRule("the-lock")
oRule {
        SetDomain("scope")
        When("kind", "equals", "world")
        When("admitted-by", "missing", "")
        ThenViolation("a world entered the cycle with no written admission")
}
g = new stzGraph("cycle-1")
g {
        AddNode("customer-phone")
        SetNodeProperty("customer-phone", "kind", "world")
        SetNodeProperty("customer-phone", "admitted-by", "the opening scope")
        AddRule(oRule)
        ? RulesAreSound()
        #--> 1
        AddNode("loyalty-club")
        SetNodeProperty("loyalty-club", "kind", "world")
        RulesReport()
        #--> Rule report 'cycle-1': 1 finding(s), 1 error(s), 0 warning(s) -> UNSOUND (errors
present)
        #      [scope] 1 finding(s)
        #        ERROR the-lock @ loyalty-club -- a world entered the cycle with no written
admission
        SetNodeProperty("loyalty-club", "admitted-by", "a notebook entry, and one exit written")
        ? RulesAreSound()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'idée d'un club de fidélité arrive, et le cycle n'est plus sain ! Le tribunal répond `UNSOUND (errors present)`, et nomme la règle et le nœud : `ERROR the-lock @ loyalty-club`. Le cycle redevient sain quand l'idée porte son admission écrite.

Essayer des paiements sans abonnement. La règle du projet était de tester sans s'engager. Cela voulait dire le bac à sable gratuit d'un établissement de paiement et les services simulés de la bibliothèque, et aucun abonnement avant une barrière que les notes appellent la barrière de mise en service. Le côté de la bibliothèque est un bac à sable de paiements qui n'est pas complaisant. Il se comporte comme une passerelle : un rejet est une réponse, une panne n'en est pas une, et une capture ne peut pas être faite deux fois. Les montants sont des entiers en unités mineures, donc 4 200 font quarante-deux euros. Que répond-il à un jeton qu'on lui a dit de rejeter, à une panne, et à une seconde capture ? Lancez-le :

HARO

```
oPay = new stzPaymentsSandbox()
oPay {
        DeclineToken("tok_chargeback")
        ? Authorize(4200, "tok_chargeback")[:status]
        #--> declined
        FailNext()
        ? Authorize(4200, "tok_visa")[:status]
        #--> refused
        aAuth = Authorize(4200, "tok_visa")
        ? aAuth[:status]
        #--> authorized
        Capture(aAuth[:id])
        ? Capture(aAuth[:id])[:why]
        #--> this authorization was already captured
}
```

✓ exécuté : a affiché ce que ce bloc promet

Trois lignes imprimées, un rejet, un refus et une raison, et aucune n'est une chaîne vide : `declined`, `refused`, et `this authorization was already captured`.

Et la barrière de production refuse de s'ouvrir tant qu'un faux est encore lié, même pour une personne habilitée. Ici le lien réel est un bac à sable HTTP qui tient la place d'un adaptateur, comme le fait la garde : le registre ne tient que le nom du secret.

Que répond la barrière de production quand le propriétaire, habilité, demande la production tant que le faux est encore lié ? Lancez-le :

HARO

```
oReg = new stzServiceRegistry("shop")
oReg.BindSandbox(:payments, new stzPaymentsSandbox())
oDel = new stzDelivery("shop")
oDel {
        AddBackend("api", "linux")
        UseServices(oReg)
        NeedsServiceIn("api", [ :payments ])
        DeployTo(StzDeploymentSiteQ("host1"), "api")
        SetActor( HumanActor("owner") )
        ? Deploy(:Production).WasCommitted()
        #--> 0
        ? Log().EntriesOfLevel(:error)[1][:message]
        #--> REFUSED -- the external surface is not production-ready
        ? WhyServicesNotReady()
        #--> sandbox-in-production: still bound to a SANDBOX in a production phase -- a fake must
never ship
        ServicesQ().BindLiveQ(:payments, new stzHttpSandbox(), "payments_key")
        ? ServicesAreProductionReady()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Le propriétaire est habilité, et il est refusé ! Rien n'est validé, et le journal dit pourquoi : `REFUSED -- the external surface is not production-ready` ; la raison dessous est la règle : `a fake must never ship`. Elle s'ouvre quand le lien est déclaré réel : la barrière lit la nature du lien, bac à sable ou réel, et non la classe derrière lui, et c'est pourquoi un bac à sable HTTP lié comme réel l'ouvre ici.

La constellation. Quand il a renversé l'ordre du travail, l'architecture du projet a été redessinée en une constellation de cinq mondes sur un socle commun. Les mondes sont le téléphone du client, l'écran de la cuisine, l'atelier du commerçant, la gestion, qui est la première application gardée et mise en sommeil, et la console de l'éditeur. Le socle commun est l'identité, le catalogue, les commandes, les paiements et la flotte. Dans la bibliothèque, le dernier monde dans l'ordre du travail peut être mis en sommeil, et il est réveillé quand son tour vient. Que répond un appel à travers un lien tant que le monde visé dort, et une fois qu'il est réveillé ? Lancez-le :

HARO

```
oCon = new stzSuperApp("neighbourhood")
oCon {
        AddWorld("customer", StzAppQ("customer"))
        AddWorld("kitchen", StzAppQ("kitchen"))
        AddWorld("supplier", StzAppQ("supplier"))
        Bond("customer", "kitchen", "send-order")
        Bond("kitchen", "supplier", "order-produce")
        GovDeclareRisk("send-order", 1)
        GovDeclareRisk("order-produce", 2)
        GovGrant("customer", "send-order")
        GovGrant("kitchen", "order-produce")
        GovSetAuthority("customer", :Delegated)
        GovSetAuthority("kitchen", :Delegated)
        Retire("supplier")
        ? CallAcross("customer", "kitchen", "send-order")
        #--> 1
        ? CallAcross("kitchen", "supplier", "order-produce")
        #--> 0
        ? Why()
        #--> target world 'supplier' is not active.
        Revive("supplier")
        ? CallAcross("kitchen", "supplier", "order-produce")
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'appel à travers le lien passe. L'appel vers le monde en sommeil est refusé ! La raison tient en une ligne : `target world 'supplier' is not active.` Réveillé, il passe.

Le budget de vitesse. La première priorité du propriétaire était la vitesse, donc la vitesse devait être une promesse jugée par quelque chose, pas une aspiration. Je tiens cela pour une vue, non pour un résultat. Le budget a trois nombres : le temps du premier affichage, le temps de réaction à un appui, et le temps pour qu'une commande arrive en cuisine. La bibliothèque les déclare à côté du code comme des attentes et les juge contre ce qui a été mesuré. Un dépassement fait échouer la même barrière qu'un constat de sécurité. Il a resserré les seuils lui-même, dans son second cadrage, et je n'en imprime aucun. Le nom du moniteur, les bornes et les échantillons de ce bloc sont les miens, pour montrer la forme ; ce ne sont pas le budget du projet. À quoi ressemble un dépassement ? Lancez-le :

HARO

```
oMon = StzPerfMonitor("remote-control")
oMon {
        NewTimer("first.display.ms").Record(900)
        NewTimer("tap.ms").Record(40)
        NewTimer("order.to.kitchen.ms").Record(1800)
}
oSla = StzSla("speed-budget")
oSla {
        ExpectP95("first.display.ms").Under(1200)
        ExpectP95("tap.ms").Under(80)
        ExpectP95("order.to.kitchen.ms").Under(2500)
        CheckAgainst(oMon)
        Show()
        #--> SLA speed-budget -- 3 expectation(s), last check: MET.
        #      [MET] p95-first.display.ms-under-1200: measured 900 (must be <= 1200)
        #      [MET] p95-tap.ms-under-80: measured 40 (must be <= 80)
        #      [MET] p95-order.to.kitchen.ms-under-2500: measured 1800 (must be <= 2500)
}
oMon.MetricQ("order.to.kitchen.ms").Record(3100)
oSla.CheckAgainst(oMon)
oRep = new stzRuleReport("cycle-gate")
oRep.Ingest(oSla.Findings())
? oRep.IsSound()
#--> 0
? oSla.Findings()[1][:message]
#--> p95-order.to.kitchen.ms-under-2500: measured 3100 (must be <= 2500)
```

✓ exécuté : a affiché ce que ce bloc promet

Le premier contrôle imprime `3 expectation(s), last check: MET.` Puis une commande prend 3 100 millisecondes, et la barrière n'est plus saine ! Le constat le dit dans les mots du budget : `p95-order.to.kitchen.ms-under-2500: measured 3100 (must be <= 2500)`.

La bibliothèque porte-t-elle ce que le projet a appris des interfaces ? Non. Le fichier de règles d'interface du projet est de l'expérience en quelques lignes, quinze règles sous une seule directive que le projet appelle le test des trois secondes, qu'un restaurateur n'a pas le temps de regarder plusieurs secondes un écran encombré : une action claire par écran ; annuler plutôt que confirmer ; un bouton refusé est désactivé à la vue, jamais caché. Et un test : une première journée simulée sur une base vide a trouvé le trou le plus grave, qu'aucun fournisseur ne pouvait être créé en production. Ces règles ont été forgées, au compte du projet lui-même, par onze

itérations et une campagne de vingt-deux tests simulés avec dix-sept corrections. La bibliothèque n'en porte aucune comme contrôle. Donc, selon la règle de ce chapitre, elles sont la leçon du projet et non celle de la bibliothèque, et je les raconte ici sans bloc.

Six refus suivent, chacun avec sa raison.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé le grand système d'un seul bloc. Après deux ans de spécification riche, le projet a failli s'arrêter. L'estimation de 2025 reste la référence pour le périmètre du projet ; elle ne doit jamais être présentée comme le calendrier d'aujourd'hui. Ce qui l'a remplacée est la doctrine de ce chapitre : livrer petit, vite et utile.

J'ai refusé de jeter la première application. Quand le propriétaire a renversé l'ordre, l'application déjà construite a été gardée, en sommeil, jusqu'à ce que la première version du nouvel ordre soit en production. Les mots du projet lui-même : rien n'est jeté.

J'ai refusé la dérive silencieuse. Le Verrou existe parce qu'un projet qui a échoué une fois dérive sans que personne ne l'ait décidé. Un second changement de cap avant que la première version tourne en production est refusé, et une idée qui entre dans le cycle sans sortie écrite l'est aussi.

J'ai refusé le faux complaisant. Un double qui répond en silence par une chaîne vide produit un test qui passe pour la mauvaise raison. Le bac à sable décline quand une passerelle décline, refuse quand une passerelle est en panne, et refuse une seconde capture. Et j'ai refusé un tarif fixe avant la barrière de mise en service : aucun abonnement à un établissement de paiement avant l'accord écrit du propriétaire.

J'ai refusé de prêter au propriétaire ce qu'une simulation avait dit. Le projet porte une règle depuis juillet 2026, selon laquelle aucune action de ses écrans ne prend plus de deux gestes. Le site l'imprime comme une part du premier avenant, et les documents du projet la portent comme une contrainte du propriétaire. Sa plus ancienne source est une ligne du fichier de réponses simulées du 3 juillet, écrite par moi et une session. Le projet se l'est donnée à lui-même, à partir de sa propre simulation de ses réponses, et l'a gardée. Je la raconte ainsi. Ce qu'il a dit, lui, et que je tiens pour sien, c'est la priorité de la vitesse.

J'ai refusé de dire que la bibliothèque était née de son besoin. Voici les dates. La conception de la constellation, le plan de virtualisation, le registre, le port des paiements et le jugement de performance ont tous été commités pour la première fois après le renversement du 4 juillet. Le plan est daté du 23 juillet, et l'étude de performance du 29 juillet. Le site dit avec soin que le projet avait besoin de ces choses et que la bibliothèque les porte. Je n'en dirai pas plus. Le dossier ne dit pas ce qui est venu en premier, son besoin ou les parties : il montre le projet les trouvant dans la bibliothèque, et je laisse l'ordre ouvert.

Cinq gardes se tiennent derrière les cinq blocs ci-dessus.

5. LE TRIBUNAL

`06_stzsuperapp_narrated` prouve la constellation : des mondes qui se joignent, un appel entre mondes qui exige que les deux soient actifs, un lien et une gouvernance déclarés, et un monde retiré qui rompt ses liens puis se réveille. `service_delivery_narrated` prouve qu'un déploiement en production par un acteur habilité est refusé tant qu'un faux est lié, chaque raison étant consignée, et qu'il s'ouvre une fois le lien réel. `payments_port_narrated` prouve la machine à états du bac à sable : capturer une fois, capturer seulement un débit autorisé, ne pas rembourser plus que ce qui a été capturé. Un rejet est une réponse et une panne n'en est pas une. `judgment_narrated` prouve une attente déclarée dans le vocabulaire opérationnel et jugée contre des nombres mesurés, un dépassement comme un constat qui fait échouer la barrière. Une attente sur un nombre que personne ne mesure y est une erreur, et jamais un passage silencieux. `graphrule_object_narrated` prouve la règle comme un objet attaché à un modèle, avec les opérateurs qu'utilise la construction du Verrou.

Le tribunal prouve ce que la bibliothèque fait ; il ne dit pas où le terrain fait mieux.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`06_stzsuperapp_narrated` 21 assertions
base/test/app/06_stzsuperapp_narrated.ring

`service_delivery_narrated` 60 assertions
base/test/system/service_delivery_narrated.ring

`payments_port_narrated` 49 assertions
base/test/system/payments_port_narrated.ring

`judgment_narrated` 41 assertions
base/test/perf/judgment_narrated.ring

`graphrule_object_narrated` 34 assertions
base/test/graph/graphrule_object_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en stockage, en requêtes et en écosystème. L'étude propre de la bibliothèque sur le terrain de la performance, écrite contre les outils du début de 2026, le dit sans l'adoucir. Pas de base de données de séries temporelles, pas de langage de requête pour elle, pas de tableaux de bord, par conception ; l'export à la place. Parmi les cinq propriétés qu'elle revendique contre eux, celle que ce chapitre utilise est la première : rien sur le terrain ne couple un budget au propre système de règles du code.

Le terrain mène dans le navigateur. Les mondes du restaurant tournent dans les navigateurs des clients et de la cuisine, et je n'ai rien trouvé dans la bibliothèque qui y mesure un premier affichage ou un appui. Le moniteur de la bibliothèque observe un serveur, et le SLA juge les nombres qu'on enregistre dans un chronomètre. Le

protocole de mesure du propriétaire est au soixante-quinzième percentile sur un téléphone de milieu de gamme. Les termes de la bibliothèque ne peuvent pas le déclarer exactement : ils offrent le cinquantième, le quatre-vingt-quinzième et le quatre-vingt-dix-neuvième.

Le terrain mène sur les services réels. Un double n'est pas le service, et le plan le dit : il faut encore au moins une exécution réelle avant la production, et les réponses enregistrées dérivent. Les vrais fournisseurs mènent dans les cas limites que le bac à sable ne modélise pas.

Et les hôtes de super-applications mènent en base installée. La conception nomme ce que la super-application de l'industrie possède, une boutique, une revue, un pont natif, et le remplace ; elle ne revendique pas d'utilisateurs.

Je dis maintenant où le chapitre s'arrête.

7. LA LIMITE

CONSTRUIT — la constellation de mondes avec ses liens et ses normes ; le bac à sable des paiements et la barrière de production qui refuse un faux ; le budget de vitesse déclaré et jugé

CONÇU — le Verrou comme règle de la bibliothèque ; la constellation écrite dans l'idiome proche du naturel que la conception décrit

ABSENT — l'adaptateur de paiement réel du restaurant ; une mesure du budget sur un vrai réseau mobile ; une attente de vitesse dans le percentile propre au propriétaire ; la mesure du temps côté navigateur ; la plateforme sur le moteur Softanza

Ce qui est construit : la constellation de mondes avec ses liens et ses passages gouvernés par des normes, le retrait, le réveil, l'échange à chaud et l'imbrication ; le bac à sable des paiements et la barrière de production. Le budget de vitesse déclaré et jugé ; les règles de graphe qu'utilise la construction du Verrou.

Ce qui est conçu : le Verrou comme règle de la bibliothèque, dont le dossier n'a aucun code ; l'idiome proche du naturel de la constellation, avec les verbes que la conception nomme et que la classe n'a pas.

Ce qui est absent : l'adaptateur de paiement réel du restaurant, que la charte de la bibliothèque appelle une tâche ultérieure et la décision du propriétaire. Une mesure du budget sur un vrai réseau mobile, que le tableau de bord du projet du 23 août a reportée au cycle suivant. Les règles d'interface, dont la règle des deux gestes, comme contrôles, une attente au soixante-quinzième percentile et la mesure du temps côté navigateur, dont aucun document de la bibliothèque ne porte la conception. Et l'état de la plateforme après le 25 août 2026, où le dossier s'arrête. Rien n'y dit que la première version soit arrivée en production au restaurant, ait passé la barrière de mise en service, ou ait pris une vraie commande d'un vrai client. Le dossier est muet sur ce qui s'est passé après le 25 août 2026, et je le dis. Des seuils de vitesse resserrés du propriétaire, je n'en imprime aucun, et dis seulement qu'un tel budget existe.

Et ce qui tourne aujourd'hui est ordinaire. Les mondes sont des fichiers HTML uniques, un par monde, avec un cadriciel embarqué, sans étape de construction et sans réseau de contenu. Le serveur est un programme Node de quelques centaines de lignes. Il utilise cinq modules propres à Node, sert les mondes sur le réseau local, et ne garde qu'une chose : un journal, chaîné par des empreintes, rejoué au redémarrage. La première application de gestion a une enveloppe de téléphone autour d'une vue web. Rien de tout cela ne tourne sur le moteur Softanza. La bibliothèque est la référence de conception du projet. La constellation est, selon les mots du site, une conception que la bibliothèque sait exécuter. Le restaurant, non.

Six portes suivent, une pour chaque lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Vous déclarez des mondes et des liens et vous exécutez une constellation où un appel vers un monde qui n'est pas encore actif est refusé avec sa raison. Le code du restaurant, lui, est une application web.
L'analyste	La rigueur dans les nombres était le contrat de la première application : une échéance calculée à partir de la date de facture, une date obligatoire, une dette sans échéance rendue visible.
Le concepteur	Quinze règles d'interface à partir d'une seule directive, le test des trois secondes ; et un bouton refusé est désactivé et visible, jamais caché. Sur qui tenait le téléphone dans les onze itérations, les fichiers du projet disent les deux, l'utilisateur réel et moi, et le chapitre ne tranche pas.
Le décideur	Le propriétaire a accepté le Verrou par écrit, a resserré lui-même le budget de vitesse, et a gardé la barrière sur les tarifs fixes. Le miroir de la bibliothèque est un déploiement qu'une personne habilitée peut valider et qui est quand même refusé tant qu'il porte un faux.
Le formateur	Le dossier ne soutient qu'une petite chose : la première application enseignait son utilisateur en faisant, avec des guides dont les mots ne laissent aucun doute. Il n'y a pas de formateur dans cette histoire, et je le dis.
L'agent	Un agent d'IA peut conduire un déploiement en production sans rien changer : la barrière laisse passer les services, rien n'est validé, et la raison donnée est l'acteur et non les services. Dans le projet, l'agent d'IA est la session qui travaille à mes côtés sous le Verrou, avec une personne qui décide.

9. EXERCICE

Déclarez votre propre Verrou pour un cycle. Puis écrivez les mondes du cycle où vous êtes, trois d'entre eux, comme le ferait un restaurant : le téléphone du client, l'écran de la cuisine, et un de plus, chacun avec la raison écrite de sa présence. Ajoutez-en un quatrième, l'idée que vous avez eue ce matin. Lancez le tribunal. Changez la déclaration, pas le tribunal, jusqu'à ce qu'il dise que le cycle est sain.

HARO

```
? YourGraph.RulesAreSound()
#--> 1
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Le tribunal vous dira la circonstance et la règle. Il n'écrira pas la déclaration. Si vous voulez une épreuve plus dure, ajoutez une règle sur les actions de votre cycle, et trouvez l'action qui coûte un troisième geste.

Je vous laisse celui-ci, volontairement. Le protocole de vitesse du propriétaire est au soixante-quinzième percentile, et le budget de la section 3 est jugé au quatre-vingt-quinzième. Prenez ce bloc, changez le percentile d'une attente pour un autre que la bibliothèque offre, gardez toutes les autres lignes, et lancez-le. Lisez ce que le tribunal juge contre ce qu'il a demandé. La section 6 dit ce que les termes de la bibliothèque offrent, et la section 7 range son attente parmi les choses absentes. Je ne vous donne ni le verdict ni la réponse.

Quand vous l'aurez résolu, vous saurez pourquoi une promesse que le tribunal sait juger n'est pas encore la promesse que le propriétaire a demandée.

Le chapitre 29 est une banque à Niamey, où une organisation est jugée par son propre organigramme, selon les règles d'un régulateur, par un programme qui lit l'organigramme comme une donnée. L'organigramme dit ce qui est, la règle dit ce qui doit être, et le verdict nomme la règle et le lieu.

OÙ SONT LES PREUVES

NARRATION	stz-service-virtualization-code-first-subscribe-later.md
NARRATION	stz-perf-judgment-narration.md
NARRATION	stzApp-building-your-first-app-narration.md
SUR LE SITE	forged
SUR LE SITE	customers
DÉCISIONS DE L'ÉTUDE	12.15, 12.19, 12.25, 12.29, 12.30

PARTIE VI · CHAPITRE 29

Une banque à Niamey

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 6 bloc(s) sur 6 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends une banque de Niamey, que j'appelle Softibank (le nom est fictif), sous la banque centrale de l'union monétaire ouest-africaine, et je dis d'abord où est la leçon : dans la bibliothèque, et non dans le produit que la banque fait tourner. J'écris l'organigramme d'une petite banque comme un texte et je le juge selon la règle régionale et selon la séparation des tâches. J'imprime ce que chacun des sept régimes contient, j'ajoute une règle propre à la banque comme une donnée, et je lis le rapport unique avec son verdict. Je dis les quatre choses que j'ai refusées, la première étant de cacher la structure dans une base de données. Le principe tient en une phrase : l'organigramme dit ce qui est, la règle dit ce qui doit être, et le verdict nomme la règle et le lieu.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Une banque est jugée sur sa structure autant que sur ses chiffres. Un régulateur demande qui rend compte à qui : si la fonction d'audit est indépendante des dirigeants qu'elle audite, s'il y a un conseil, si une fonction de risque existe. Il demande si la personne qui exécute un paiement est celle qui l'approuve. Le terrain répond par des documents : un organigramme dessiné dans une diapositive, une politique écrite dans un traitement de texte, un audit qui compare les deux une fois par an.

Le terrain a aussi une famille d'outils appelés plateformes de gouvernance, de risque et de conformité, qui tiennent des registres de risques et de contrôles et suivent les constats jusqu'à leur clôture. Ils enregistrent qu'un contrôle existe et à qui il appartient. Ils ne lisent en général pas la structure de l'organisation pour vous dire qu'elle est fausse. C'est une connaissance générale, pas une étude du dossier.

2. LE PREMIER PRINCIPE

Une organisation peut être jugée par son propre organigramme, selon les règles d'un régulateur, par un programme qui lit l'organigramme comme une donnée : l'organigramme dit ce qui est, la règle dit ce qui doit être, et le verdict nomme la règle et le lieu.

3. LA FORME SOFTANZA

Une banque commerciale de Niamey, sous les règles de la banque centrale de l'union monétaire ouest-africaine, fait tourner un produit d'évaluation des organisations que j'ai construit. Il est installé sur le réseau interne de la banque, sous licence, et le guide d'installation que j'ai remis à la banque est le document qui atteste la référence. Le produit lui-même est une application Python : un petit serveur web avec une base de données. Je le dis d'abord, parce que la leçon que je vais vous montrer est dans la bibliothèque, et non à l'intérieur de ce produit.

La leçon est qu'un organigramme est une donnée qu'un programme peut juger. Elle commence par un texte qu'une personne peut éditer, sans programmer : des postes, leurs titres, leurs départements, et à qui chacun rend compte. Voici l'organigramme d'une petite banque. Que dit de lui la règle du régulateur régional ? Lancez-le :

HARO

```
cChart = '
orgchart "bank"

positions
    board
        title: Board of Directors
        level: executive
        department: board
        reportsTo:
```

```
    ceo
        title: Chief Executive
        level: executive
        department: executive
        reportsTo: board

    risk
        title: Chief Risk Officer
        level: management
        department: risk
        reportsTo: board

    audit
        title: Internal Audit
        level: management
        department: audit
        reportsTo: ceo
'
oOrg = new stzOrgChart("bank")
oOrg {
        ImportStzOrg(cChart)
        ? NodeCount()
        #--> 4
        ? @@( ValidateXT(:BCEAO)[:issues] )
        #--> [ "BCEAO-002: Audit reports to non-board position" ]
        ? IsValidXT(:BCEAO)
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

La règle du régulateur lit l'organigramme et le refuse ! L'audit rend compte au directeur général, et le verdict le dit dans ses propres mots, `BCEAO-002: Audit reports to non-board position`, avec un code de règle qu'un auditeur peut retrouver. Un organigramme est jugé par plus d'un tribunal à la fois, chaque constat porte son propre code, et le rapport de conformité dit conforme ou non. Que dit un second tribunal, la séparation des tâches, qui veut que les opérations ne rendent pas compte à la trésorerie ? J'ajoute la trésorerie et les opérations, les opérations sous la trésorerie, et je lance les deux tribunaux :

HARO

```
oOrg {
        AddManagementPositionXT("treasury", "Treasury")
        SetPositionDepartment("treasury", "treasury")
        AddManagementPositionXT("ops", "Operations")
```

```
        SetPositionDepartment("ops", "operations")
        ReportsTo("treasury", "ceo")
        ReportsTo("ops", "treasury")
        aV = ValidateXT([ :BCEAO, :SOD ])
        ? aV[:validatorsFailed]
        #--> 2
        ? aV[:results][1][:issues][1]
        #--> BCEAO-002: Audit reports to non-board position
        ? aV[:results][2][:issues][1]
        #--> SOD-001: Operations reports through Treasury
        ? ComplianceReport()[:overallStatus]
        #--> non-compliant
}
```

✓ exécuté : a affiché ce que ce bloc promet

Deux validateurs échouent, chacun avec son propre code, et le rapport de conformité dit `non-compliant`.

La bibliothèque nomme sept régimes, par le nom de leurs classes : le régional, Bâle III, SOX, PCI DSS, ISO 27001, le RGPD et HIPAA. Ils partagent un seul mécanisme. Combien de règles chacun contient-il ? Je l'imprime, plutôt que d'écrire une phrase qui dit « Bâle III est validé » :

HARO

```
aBases = [ new stzBCEAORuleBase(), new stzBaselIIIRuleBase(), new stzSOXRuleBase(), new
stzPCIDSSRuleBase(),
          new stzISO27001RuleBase(), new stzGDPRRuleBase(), new stzHIPAARuleBase() ]
for oBase in aBases
       ? oBase.Name() + " : " + oBase.NumberOfRules()
next
#--> BCEAO : 4
#--> Basel III : 4
#--> SOX : 5
#--> PCI-DSS : 4
#--> ISO 27001 : 4
#--> GDPR : 4
#--> HIPAA : 4
```

✓ exécuté : a affiché ce que ce bloc promet

Six régimes portent quatre règles, et un en porte cinq ! Quatre règles que chaque base porte : personne ne rend compte à soi-même, aucun cycle de rattachement, aucun poste orphelin, et une étendue de contrôle. SOX en ajoute une qui lui est propre : aucun poste ne tient à la fois le rôle d'approbateur et celui d'exécutant. Elle est appelée aussi séparation des tâches, mais c'est une règle différente du second tribunal ci-dessus. Le commentaire de la bibliothèque le dit sans détour : les classes de régimes ne diffèrent aujourd'hui que par le nom, et une seule ajoute une règle qui lui est propre. Les quatre sont celles de la base de règles ; le validateur qui a refusé l'organigramme dans le premier bloc tient trois règles qui lui sont propres, codées BCEAO-001 à BCEAO-003, et les deux ne sont pas la même chose. Je reviendrai sur ce que cela veut dire.

La règle propre d'une banque est aussi une donnée. Prenez la règle selon laquelle l'audit interne rend compte au conseil. C'est un objet déclaré par des appels. Elle nomme le département qu'elle gouverne, la ligne qu'elle exige et la phrase qu'elle imprime, et elle s'ajoute à la base de règles d'un régime. Quelle règle a été rompue, et où ? Le rapport le dit :

HARO

```
oRule = new stzGraphRule("audit-reports-to-board")
oRule {
        SetDomain("orgchart")
        When("department", "equals", "audit")
        ThenQ("reportsTo", "equals", "board")
        ThenViolation("internal audit must report to the board")
}
oBasel = new stzBaselIIIRuleBase()
oBasel.AddRule(oRule)
oRep = new stzRuleReport("bank")
oRep.Ingest(oOrg.CheckCompliance(oBasel))
oRep.Report()
#--> Rule report 'bank': 1 finding(s), 1 error(s), 0 warning(s) -> UNSOUND (errors present)
#      [orgchart] 1 finding(s)
#        ERROR audit-reports-to-board @ audit -- internal audit must report to the board
```

✓ exécuté : a affiché ce que ce bloc promet

Le rapport dit `UNSOUND` et nomme la règle et le lieu : `ERROR audit-reports-to-board @ audit -- internal audit must report to the board`.

Puis le verdict suit l'organigramme. Deux modifications de deux lignes de rattachement, l'audit rend désormais compte au conseil et les opérations au directeur général. Que disent maintenant les tribunaux qui disaient non ? Lancez-le :

HARO

```
oOrg {
        ReportsTo("audit", "board")
        ReportsTo("ops", "ceo")
        ? IsValidXT([ :BCEAO, :SOD ])
        #--> 1
        ? ComplianceReport()[:overallStatus]
        #--> compliant
}
oRep = new stzRuleReport("bank")
oRep {
        Ingest(oOrg.CheckCompliance(oBasel))
        Report()
        #--> Rule report 'bank': 0 finding(s), 0 error(s), 0 warning(s) -> SOUND
        ? IsSound()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Ils disent `compliant`, et le rapport dit `SOUND`. C'est le mouvement que raconte la narration d'une banque virtuelle, d'une structure non conforme à une structure gouvernée. Le code propre de la narration ne s'exécute pas au commit épinglé ; les appels ci-dessus, si. Le rapport est une seule barrière sur les bases de règles qu'on lui donne. Et voici la limite de ce mouvement, sous forme de bloc. Un tribunal lit ce que l'organigramme déclare, et rien de plus. La règle régionale ne compare le département du supérieur d'un poste que lorsque ce département est déclaré, donc je m'attends à ce que le constat sur l'audit disparaisse quand le département du directeur général est laissé vide. Que dit-il quand il l'est ? Lancez-le :

HARO

```
oOrg2 = new stzOrgChart("bank")
oOrg2 {
        ImportStzOrg( substr(cChart, "department: executive", "department:") )
        ? IsValidXT(:BCEAO)
        #--> 1
        ? ValidateXT(:Banking)[:status]
```

```
        #--> pass
        SetPositionDepartment("ceo", "executive")
        ? IsValidXT(:BCEAO)
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

Il dit `1` : une fonction d'audit qui rend compte au directeur général passe ! Déclarez le département, et le constat apparaît. Le même bloc montre le validateur bancaire qui répond `pass` : le commentaire de la bibliothèque dit qu'il passe à chaque fois, un emplacement réservé pour des règles pas encore écrites.

Mon livre précédent, *Refinement-Oriented Programming*, raconte la discipline qui gouverne le changement à travers l'histoire de Softibank, une banque face à la lettre d'un régulateur. Ce chapitre ne la raconte pas de nouveau. Il dit ce que la bibliothèque porte.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé de cacher la structure dans une base de données. Dans le produit livré, un organigramme est des lignes dans deux tables. Dans la bibliothèque, c'est un texte qu'une personne peut lire et éditer, et il s'importe dans l'objet que le tribunal juge. Une règle appliquée à la ligne d'une base est une requête ; une règle appliquée à un texte est quelque chose qu'une personne peut lire avant qu'elle ne s'exécute.

J'ai refusé de prétendre plus que ce que la bibliothèque contient. Le mot « validateur » peut se lire pour deux choses différentes. Le régulateur régional a un validateur de trois règles, et une base de règles de quatre règles universelles. Bâle III n'a que la base de règles, et les autres aussi. J'ai imprimé le compte, et le bloc ci-dessus le montre.

J'ai refusé de porter le texte d'un régulateur dans un code dont je ne peux pas citer la source. Les codes BCEAO-001 à BCEAO-003 sont ceux de la bibliothèque. Le dossier ne les rattache à aucun article du texte de la banque centrale. Ce sont les propres règles de la bibliothèque, nommées d'après le régime du régulateur ; le dossier ne nomme aucun texte dont elles seraient tirées.

J'ai refusé de dire ce que je ne sais pas. Ce que les gens de la banque déclarent, exploitent ou disent n'est pas dans le dossier. Le guide d'installation est mon document à la banque : des étapes d'installation, un fichier de licence, un contact de support. La page Forged ne garde aucun chiffre ni aucune personne d'un client. Le dossier est muet sur qui, à la banque, déclare l'organigramme, le questionnaire ou les règles. Il est muet sur depuis combien de temps le produit est installé, et sur combien de personnes l'utilisent. Je n'en inventerai rien, et le chapitre ne comble pas le vide. Sur « construit avec Softanza », le dossier montre une application Python, un petit serveur web avec une base de données, qui n'appelle rien de la bibliothèque. Les validateurs de la bibliothèque en sont séparés, et ce sont eux que ce chapitre exécute. Le chapitre 1 dit que la construction de la plateforme a façonné Softanza ; ce chapitre dit ce que le dossier montre.

Six gardes se tiennent derrière les blocs.

5. LE TRIBUNAL

`orgrule_narrated` prouve SOX, le RGPD et HIPAA comme bases de règles sur le graphe de l'organigramme, avec les quatre règles universelles et la séparation des tâches que SOX ajoute, et l'organigramme qui rejoint le rapport unique. Le bloc ci-dessus imprime les sept comptes. `14_bceao_governance` prouve deux des trois règles du validateur régional. `15_multiple_validators` prouve le verdict combiné, et `13_segregation_of_duties` le second tribunal. `rulereport_narrated` prouve le rapport unique sur les constats du code, des agents d'IA et de la sécurité. `36_import_from_stzorg` prouve la forme texte de l'organigramme.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`orgrule_narrated` 22 assertions base/test/graph/orgrule_narrated.ring

`14_bceao_governance` 1 assertions
base/test/orgchart/14_bceao_governance.ring

`15_multiple_validators` 1 assertions
base/test/orgchart/15_multiple_validators.ring

`13_segregation_of_duties` 1 assertions
base/test/orgchart/13_segregation_of_duties.ring

`rulereport_narrated` 23 assertions
base/test/graph/rulereport_narrated.ring

`36_import_from_stzorg` 3 assertions
base/test/orgchart/36_import_from_stzorg.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en couverture. Une plateforme de gouvernance et de conformité tient en général un registre de réglementations, de contrôles et de constats, avec des flux de travail, des responsables et des échéances. C'est une connaissance générale, pas une étude du dossier. La bibliothèque tient un contrôle structurel d'un régulateur régional, un de séparation des tâches, et un mécanisme. Pour le RGPD, HIPAA et les autres, elle tient le mécanisme et les quatre règles universelles, et rien qui leur soit propre.

Le terrain mène en usage réel : une telle plateforme est souvent une que les auditeurs d'une banque connaissent déjà, et qu'un régulateur a déjà vue. C'est une connaissance générale, pas une étude du dossier.

Là où je pense que ceci se distingue, c'est que l'organigramme est la donnée et que le tribunal est une règle sur elle. Un organigramme est un texte, une règle est une donnée, un constat nomme la règle et le poste, et le rapport est un seul verdict. Le tableau de comparaison propre de la bibliothèque pour cette fonction n'est que victoires, sans dates, avec des chiffres de coût qui ne citent rien, et je ne l'imprimerai pas.

Voici la limite, dans les mots d'étape.

7. LA LIMITE

CONSTRUIT	l'organigramme comme un texte qu'une personne peut éditer ; les validateurs du régulateur régional et de la séparation des tâches ; les sept régimes comme un seul mécanisme ; une règle écrite comme une donnée ; un rapport avec un verdict
CONÇU	le contenu qui ferait différer les six autres régimes du premier ; un fichier de règles qui juge, et pas seulement qui se charge
ABSENT	le produit livré tournant sur la bibliothèque ; un mot des gens de la banque ; la lecture, par un auditeur, d'un constat produit par la bibliothèque

Ce qui est construit : l'organigramme comme modèle et comme texte ; les validateurs du régulateur régional et de la séparation des tâches ; les sept régimes comme un seul mécanisme avec l'exemple SOX. Construit aussi : une règle écrite comme une donnée, et le rapport unique avec son verdict.

Quatre choses sont conçues, ou construites en partie.

- Le contenu des six régimes autres que le régional. Cinq d'entre eux tiennent les mêmes quatre règles universelles et rien d'autre ; SOX en ajoute une cinquième.
- Un fichier de règles qui juge. La bibliothèque peut charger un fichier de règles dans un texte `when` et `then`, et celui de ses données de test charge deux règles. Le chargeur lit le nom, le type, la gravité, la fonction, les paramètres et le message

d'une règle, et aucune ligne `when` ni `then`. Une telle règle se charge et ne juge rien. Le format est admis, et la règle n'est pas appliquée.

- Un validateur lit des propriétés déclarées. Il trouve le conseil par le mot « board » dans le titre d'un poste, et il ne compare les départements que là où ils sont déclarés : le bloc de la limite de la section 3 vide celui du directeur général et le constat sur l'audit disparaît.
- Le validateur de succession signale chaque poste pourvu, ce qui est, selon le commentaire de la bibliothèque elle-même, un défaut connu. La simulation de réorganisation lève une erreur quand on lui demande de changer une ligne de rattachement, également un défaut connu. L'article qui raconte la simulation nomme une méthode qui n'existe nulle part dans la bibliothèque.

Quatre choses sont absentes.

- Le produit livré tournant sur la bibliothèque. C'est une application Python ; une recherche dans son code n'a trouvé aucun contrôle régional ou de Bâle et aucun appel à la bibliothèque.
- Un mot de la banque. Le dossier est muet.
- Un verdict perçu. Aucun document ne dit qu'un auditeur ou un superviseur ait lu un constat que la bibliothèque a produit.
- Une date de livraison. Les fichiers livrés portent des horodatages du système de fichiers, et ce ne sont pas une date de livraison.

La couche web du produit est née dans son propre dépôt : son premier commit est daté du 4 février 2026, et le texte de la constitution de l'interface a été commité pour la première fois le 8 février. Les champs de preuve de la constitution disent que cinq de ses règles ont été récoltées sur un produit de supervision bancaire. Le dossier ne dit pas que ce produit soit l'installation de la banque. Je ne dirai pas qu'il l'est.

Six portes suivent, une pour chaque lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	L'organigramme est un texte, la règle est une donnée, le tribunal est un appel. Un constat nomme la règle et le poste.
L'analyste	Vous changez une ligne de rattachement dans l'objet, ou dans le texte puis vous l'importez de nouveau, et vous relancez : le verdict suit l'organigramme.
Le concepteur	Le rapport est un seul verdict sur les bases de règles qu'on lui donne : sain ou non sain, avec chaque constat là où il a été trouvé.
Le décideur	Demandez quelles réglementations la bibliothèque contient, et demandez le compte. Sept régimes, un avec un validateur à côté de sa base de règles et un qui ajoute une règle ; les autres portent les quatre universelles.
Le formateur	La narration d'une banque virtuelle raconte le passage d'une structure non conforme à une structure gouvernée, et le livre montre ce passage avec des appels qui s'exécutent.
L'agent	Vous pouvez lire un organigramme et un rapport. Une règle que vous ajoutez est une donnée qu'une personne peut lire, et le tribunal juge l'organigramme avec elle.

9. EXERCICE

Dessinez l'organigramme d'une organisation que vous connaissez comme un texte de postes, de titres, de départements et de lignes de rattachement. Avant de lancer quoi que ce soit, écrivez ce que vous attendez que la règle régionale, puis la séparation des tâches, en disent. Puis écrivez une règle à vous comme une donnée, qui gouverne un département, et dites ce que le rapport imprimera.

Lancez-la. Puis laissez un département vide et dites quel constat disparaît, et pourquoi c'est dangereux.

Je vous laisse celui-ci, exprès. Prenez le bloc de la limite de la section 3 et changez une ligne : l'import prend l'organigramme entier, avec le département du directeur général déclaré.

HARO

```
oOrg2 = new stzOrgChart("bank")
oOrg2.ImportStzOrg(cChart)
? oOrg2.IsValidXT(:BCEAO)
```

```
? oOrg2.ValidateXT(:Banking)[:status]
oOrg2.SetPositionDepartment("ceo", "executive")
? oOrg2.IsValidXT(:BCEAO)
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez-le et mettez les trois réponses côte à côte. Le tribunal régional a déjà refusé cet organigramme, à la section 3 ; lisez maintenant ce que le second validateur en dit, et pourquoi ce n'est pas la même chose que ce que le tribunal a dit. Les sections 3 et 4 disent pourquoi. Je ne vous donne pas la réponse ici. Quand vous l'aurez, vous saurez pourquoi un validateur qui répond `pass` n'a pas jugé l'organigramme.

Le chapitre suivant quitte la banque pour une organisation sur le terrain, et son principe tient en une phrase : l'agent d'IA propose, et une personne décide.

OÙ SONT LES PREUVES

NARRATION	`stzOrgChartDoc-BankUseCase.md`
NARRATION	`stz-rules-as-graph-rules-narration.md`
SUR LE SITE	forged
SUR LE SITE	customers
DÉCISIONS DE L'ÉTUDE	12.19, 12.29, 12.30

PARTIE VI · CHAPITRE 30

Une organisation sur le terrain

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 7 bloc(s) sur 7 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je prends l'étude de conception d'une organisation et je montre les quatre choses qu'elle contient et que la bibliothèque porte. Ce sont une règle comme un objet dont les nombres peuvent être changés, un refus qu'une personne peut lire, l'autorité mise face au risque, et une décision gardée avec son autorité du moment. Puis je montre la chose que l'organisation a demandée et que la bibliothèque ne porte pas, le hors ligne, et je lis ce bloc comme une limite et non comme une victoire. Je dis « étude de conception » et non « livraison » chaque fois, parce que le hub n'est pas construit. Le principe est celui de l'organisation : l'agent d'IA propose, et une personne décide.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a deux pratiques dont ce projet avait besoin, et il les garde en des lieux séparés.

La première est la règle métier qu'une personne du métier peut changer. Les moteurs de règles et les tables de décision existent depuis des décennies ; le Decision Model and Notation, une norme de 2015, donne aux analystes métier une table où un plafond d'approbation est un nombre dans une cellule. La seconde est l'application qui continue de marcher sans réseau. La conception hors ligne d'abord est une pratique à part entière, avec des bases de données qui se synchronisent.

Depuis un essai de 2019, elle a un nom pour sa philosophie, le logiciel local d'abord (local-first software) : les données sont sur votre appareil, et le réseau est une commodité.

Une troisième pratique est venue avec les agents d'IA : un modèle d'IA propose, et une personne approuve. Sur le terrain, c'est un bouton.

L'étude de l'organisation fait de cette troisième pratique un principe.

2. LE PREMIER PRINCIPE

L'étude de l'organisation sur son propre projet en fait un titre de section, « L'agent IA propose, une personne décide », et je le prends comme le principe de ce chapitre : **l'agent d'IA propose, et une personne décide.**

3. LA FORME SOFTANZA

Une organisation travaille au Niger, avec un siège à Niamey et des bases sur le terrain. Sa propre spécification préférait une plateforme low-code construite sur la suite bureautique d'un grand éditeur. Mon offre du 5 août 2026 a été celle retenue. L'étude liste quatre gains, et la propriété est le premier qu'elle liste. L'organisation possède son vocabulaire et ses règles ; elle les change sans programmer ; elle ne paie aucune licence par poste pour la plateforme ; et elle peut tout confier à un programmeur de son propre pays. Nous avons construit l'étude de conception ensemble, avec ses équipes, du 22 septembre au 8 octobre 2026. Le hub n'est pas construit. La version 1.2 de l'étude est soumise au directeur exécutif pour validation, et le dossier ne dit pas qu'elle l'ait été.

Quatre choses de cette étude, la bibliothèque les porte. Je les montrerai dans l'ordre où l'organisation les rencontrerait, puis celle qu'elle a demandée et que la bibliothèque ne porte pas.

Une règle est un objet dont les nombres peuvent être changés. Un plafond d'approbation est une phrase avec un nombre dedans, attachée au modèle des avances qu'elle gouverne. Le modèle se juge lui-même, et dit combien de sujets la règle regarde. Les données ci-dessous sont inventées pour ce chapitre, quatre avances avec un montant et un compte d'approbations. Ce ne sont pas les chiffres de

l'organisation, et les nombres cinq cents et deux cents sont les miens, pas ses seuils. Un plafond de cinq cents repère-t-il une avance, et combien d'avances la règle regarde-t-elle ?

HARO

```
oG = new stzGraph("advances")
oG {
        AddNodeXTT("a0", "A0", [ :amount = 120, :approvals = 1 ])
        AddNodeXTT("a1", "A1", [ :amount = 250, :approvals = 1 ])
        AddNodeXTT("a2", "A2", [ :amount = 600, :approvals = 1 ])
        AddNodeXTT("a3", "A3", [ :amount = 900, :approvals = 2 ])
}
oRule = new stzGraphRule("second-approval")
oRule {
        SetDomain("advances")
        SetMessage("above the ceiling an advance needs two approvals")
        When("amount", "greaterthan", 500)
        ThenQ("approvals", "greaterthan", 1)
        ThenViolation("two approvals needed")
}
oG.AddRule(oRule)
oG.RulesReport()
#--> Rule report 'advances': 1 finding(s), 1 error(s), 0 warning(s) -> UNSOUND (errors present)
#      [advances] 1 finding(s)
#        ERROR second-approval @ a2 -- two approvals needed
? len(oRule.SubjectsIn(oG))
#--> 2
```

✓ exécuté : a affiché ce que ce bloc promet

Le rapport nomme une avance, `a2`, à 600 avec une seule approbation, et la règle en regarde 2 sur les quatre, `a2` et `a3`.

Que se passe-t-il quand le plafond est changé et que le même modèle est jugé de nouveau ? Ici le changement est la règle reconstruite dans le code avec un autre nombre, et le verdict et sa portée suivent :

HARO

```
oG.ClearAttachedRules()
oRule = new stzGraphRule("second-approval")
oRule {
        SetDomain("advances")
        SetMessage("above the ceiling an advance needs two approvals")
        When("amount", "greaterthan", 200)
```

```
        ThenQ("approvals", "greaterthan", 1)
        ThenViolation("two approvals needed")
}
oG.AddRule(oRule)
oG.RulesReport()
#--> Rule report 'advances': 2 finding(s), 2 error(s), 0 warning(s) -> UNSOUND (errors present)
#      [advances] 2 finding(s)
#        ERROR second-approval @ a1 -- two approvals needed
#        ERROR second-approval @ a2 -- two approvals needed
? len(oRule.SubjectsIn(oG))
#--> 3
```

✓ exécuté : a affiché ce que ce bloc promet

Le rapport dit maintenant `2 finding(s), 2 error(s)` là où il en disait un, et la portée est 3 là où elle était 2 ! Je dois être honnête sur ce que ce bloc montre. Qu'un changement de la règle soit un changement de ses valeurs, que la personne même de l'organisation fait comme on remplit un formulaire, est la promesse de l'étude. Je n'ai pas trouvé de fichier de la bibliothèque qui la réalise. La section 7 dit ce qui existe et ce qui n'existe pas.

Ce que la bibliothèque porte, c'est la réponse à la pire chose qui puisse arriver quand une personne qui ne programme pas édite une règle : une règle qui ne gouverne rien, en silence. Je tiens cela pour une vue, non pour un résultat. Faites une faute de frappe sur une propriété, et la règle ne trouve rien, et le modèle déclare qu'il est sain. Que dit un gouverneur de règles de la même règle ? La classe est `StzRuleGovernance` ; elle demande aux règles ce qu'elles gouvernent et non ce qu'elles disent, et elle nomme la règle morte :

HARO

```
oTypo = new stzGraphRule("second-approval-typo")
oTypo {
        SetMessage("above the ceiling an advance needs two approvals")
        When("amont", "greaterthan", 200)
        ThenQ("approvals", "greaterthan", 1)
        ThenViolation("two approvals needed")
        ? len(Check(oG))
        #--> 0
}
oWatch = StzRuleGovernance("hub")
oWatch {
        AddRule(oTypo)
        AddCase("advances", oG)
```

```
        aF = CheckRules()
        ? aF[1][:rule]
        #--> scope_empty
        ? aF[1][:message]
        #--> governs nothing in 1 pictures -- a dead rule and a misspelt scope predicate both
report as passing
}
oWatch2 = StzRuleGovernance("hub")
oWatch2 {
        AddRule(oRule)
        AddCase("advances", oG)
        ? len(CheckRules())
        #--> 0
}
```

✓ exécuté : a affiché ce que ce bloc promet

La règle mal orthographiée ne trouve rien, `0`, et pour le modèle, rien trouvé veut dire sain ! Le gouverneur répond `scope_empty`, et dit pourquoi : `governs nothing in 1 pictures -- a dead rule and a misspelt scope predicate both report as passing`. Une règle morte et un feu vert produisent la même sortie : le silence. Le tribunal doit distinguer « rien n'était faux » de « rien n'a été regardé ». Le mot « pictures » dans le message est l'origine du gouverneur de règles : il a d'abord été construit pour les dessins du volet des graphes, et je dirais « cas ».

Une personne peut lire le refus. Quand une réponse rompt une loi que l'organisation a déclarée, elle est refusée avec sa raison. Le refus est gardé comme un point de contrôle qu'une personne peut lire, avec son contexte. Le chapitre 17 a montré la moitié qui demande de cette machine. Voici la moitié qui refuse. Une demande a besoin d'une approbation, et la loi dit qu'une demande a un approbateur. Que fait la machine d'un second approbateur ?

HARO

```
oKB = new stzKnowledgeGraph("requests")
oKB {
        AddConversationQ("review").SetGoal(StzGoalQ().RequireOne("request-7", "approved-by"))
        ? AskIn("review")
        #--> What does 'request-7' have for 'approved-by'?  (why: 'request-7' needs 'approved-
by')
        ConstrainRelation("approved-by", :Unique)
        KnowRelation("request-7", "approved-by", "approver-1")
        aR = ReplyIn("review", "approver-2")
        ? aR[:refused][1][2]
```

```
        #--> 'request-7' already bears 'approved-by' (to 'approver-1'); the relation is :Unique
        aCp = ConversationQ("review").Checkpoints()
        ? len(aCp)
        #--> 1
        ? aCp[1][:attempted]
        #--> approver-2
        ? aCp[1][:why]
        #--> 'request-7' already bears 'approved-by' (to 'approver-1'); the relation is :Unique
}
```

✓ exécuté : a affiché ce que ce bloc promet

La machine demande d'abord : `What does 'request-7' have for 'approved-by'? (why: 'request-7' needs 'approved-by')`. Puis le second approbateur est refusé ! Le point de contrôle garde ce qui a été tenté, `approver-2`, et la raison, dans les mots mêmes du refus : `'request-7' already bears 'approved-by' (to 'approver-1'); the relation is :Unique`.

L'autorité n'est pas la permission. L'assistant de cette organisation peut préparer un dossier. Il peut détenir la permission de débloquer une avance et être quand même refusé, parce que son autorité ne couvre pas le risque de l'acte. Une personne qui a l'autorité est autorisée. L'assistant peut-il débloquer l'avance, et l'approbateur le peut-il ? Le verdict dit pourquoi, dans les deux cas :

HARO

```
oGov = new stzGovernance("hub")
oGov {
        DeclareRisk("prepare-file", 1)
        DeclareRisk("release-advance", 3)
        GrantPermission("assistant", "prepare-file")
        GrantPermission("assistant", "release-advance")
        SetAuthority("assistant", :Advisory)
        GrantPermission("approver-1", "release-advance")
        SetAuthority("approver-1", :Autonomous)
        ? MayProceed("assistant", "prepare-file")
        #--> 1
        ? MayProceed("assistant", "release-advance")
        #--> 0
        ? Why()
        #--> refused: 'assistant' holds 'advisory' authority (level 1) but 'release-advance' is
risk tier 3 (SHOULD does not cover it)
        ? MayProceed("approver-1", "release-advance")
        #--> 1
        ? Why()
```

```
        #--> allowed: permission held AND 'autonomous' authority (level 3) covers risk tier 3
}
```

✓ exécuté : a affiché ce que ce bloc promet

L'assistant détient la permission de `release-advance`, et la réponse est `0` ! Le verdict dit `refused: 'assistant' holds 'advisory' authority (level 1) but 'release-advance' is risk tier 3 (SHOULD does not cover it)`. Pour l'approbateur, il dit `allowed: permission held AND 'autonomous' authority (level 3) covers risk tier 3`. Dans le message, SHOULD est le mot de la bibliothèque pour ce qu'un niveau d'autorité couvre, face à CAN, la permission détenue ; la garde derrière ce bloc le dit ainsi : « pouvoir n'est pas devoir ».

La classe d'ici, `stzGovernance`, juge l'autorité ; ce n'est pas le gouverneur de règles de la règle morte plus haut. La bibliothèque ne connaît un acteur que par un nom et un niveau d'autorité. Qu'un acteur soit une personne ou un assistant, c'est ce que son niveau est déclaré être. Lier un niveau à une personne connectée est le travail du volet d'authentification, que je n'ai pas lu pour ce chapitre.

Une décision garde son autorité du moment. Chaque décision enregistre sa raison, son acteur, l'acte, l'autorité et le risque de cet instant. Une nouvelle décision remplace l'ancienne sans l'effacer, jusqu'à une capacité (512 décisions par défaut), et la gouvernance compte ce qu'elle a abandonné. Si l'on enregistre deux décisions sur le même cas, à une heure d'écart, laquelle la filiation donne-t-elle, et la première survit-elle ?

HARO

```
nAt = 1785700000000
oGov {
        RecordDecisionAt("c-1", "ceiling raised for the harvest season", "approver-1", "release-
advance", nAt)
        RecordDecisionAt("c-1", "ceiling restored, the season is over", "approver-1", "release-
advance", nAt + 3600000)
        ? LineageOf("c-1")[:rationale]
        #--> ceiling restored, the season is over
        ? len(LineageHistoryOf("c-1"))
        #--> 2
        ? LineageHistoryOf("c-1")[1][:rationale]
        #--> ceiling raised for the harvest season
        ? LineageOf("c-1")[:authorityAtTime]
```

```
        #--> autonomous
}
```

✓ exécuté : a affiché ce que ce bloc promet

La filiation répond `ceiling restored, the season is over`. Elle garde 2 décisions, et la première dit encore `ceiling raised for the harvest season`. L'autorité du moment est `autonomous`. L'instant est un nombre fixe, donc l'horloge n'est pas lue. Cela rend le bloc répétable. Ce n'est pas un magasin de versions de la règle elle-même, et je ne ferai pas semblant que ce l'est.

Et celle qu'elle a demandée et que la bibliothèque ne porte pas. Son option par défaut est de travailler hors ligne : des bases sur le terrain sans réseau, qui saisissent ce qu'elles font et l'envoient plus tard. La bibliothèque connaît le hors ligne comme une capacité déclarée et gouvernée d'un monde, demandée puis accordée ou refusée comme la caméra ou les paiements. Le hors ligne est-il accordé avant que le monde le demande, et après ? C'est tout ce que la bibliothèque en connaît :

HARO

```
oApp = StzAppQ("field-visits")
oPlat = StzPlatformQ("field")
oPlat.SetWorld(oApp)
oRegime = new stzGovernance("field-regime")
oRegime.GrantPermission("field-visits", "use-offline")
oRegime.SetAuthority("field-visits", :Advisory)
oPlat {
        SetGovernedBy(oRegime)
        ? Granted(:offline)
        #--> 0
        ? Why()
        #--> capability 'offline' was never requested via AddCapabilityQ().
        AddCapabilityQ(:offline).SetPurposes([ :enter_without_network ])
        ? Granted(:offline)
        #--> 1
        ? Why()
        #--> allowed: permission held AND 'advisory' authority (level 1) covers risk tier 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

Avant la demande, `Granted(:offline)` vaut `0`, et la raison imprimée est `capability 'offline' was never requested via AddCapabilityQ().`. Après la demande, il vaut `1`, et la raison imprimée est `allowed: permission held AND 'advisory' authority (level 1) covers risk tier 1`.

Lisez ce bloc comme une limite et non comme une victoire. Rien n'y met quoi que ce soit en file. La bibliothèque ne garde pas d'entrées sur un appareil, ne les envoie pas quand le réseau revient, et ne règle pas les modifications de deux personnes.

La section suivante pose la décision, et ce que j'ai refusé.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé une plateforme à licence par poste comme socle de l'organisation. La spécification préférait le low-code sur la suite d'un éditeur. L'offre retenue propose Softanza, et parmi ses gains l'étude met la propriété en premier. L'organisation possède son vocabulaire et ses règles, qu'elle peut changer sans programmer, et l'ensemble du travail, programmes, configuration et données, est le sien, avec une passation à un programmeur local. La suite de l'éditeur reste là où elle était bonne, pour les comptes d'utilisateurs, les documents et les notifications. Le dossier a le résultat et les gains, et non la délibération. Qui a choisi, et ce qu'on a pesé contre le low-code, il ne le dit pas. Le chapitre ne comble pas le vide.

J'ai refusé de laisser l'assistant décider. Il propose. Son autorité est consultative ; celle d'une personne couvre le risque. Le refus nomme le niveau, le niveau de risque, et la phrase qui dit que l'autorité ne couvre pas l'acte : `SHOULD does not cover it`.

J'ai refusé une règle silencieuse. Une règle qui ne gouverne rien est signalée. Une règle sans message l'est aussi. C'est le prix de laisser une personne qui ne programme pas toucher à une règle. Le gouverneur de règles existe pour ce cas, et le commentaire de la bibliothèque dit pourquoi : une règle morte et un prédicat mal orthographié produisent la même sortie, le silence.

J'ai refusé d'appeler le hors ligne une leçon. La page Forged, lue le 9 octobre 2026, dit : « C'est un manque, pas une leçon apprise. » Un restaurant a demandé la même chose. L'étude est claire sur le fait que le hors ligne est l'option par défaut de l'organisation, et la bibliothèque n'en a qu'une permission. Un paquet frère du domaine de projets, côté navigateur, contient une boîte d'envoi durable sur l'appareil et une relecture ordonnée et idempotente ; il est extérieur à la bibliothèque. Je ne l'ai pas exécuté.

Six gardes se tiennent derrière les blocs.

5. LE TRIBUNAL

`graphruleset_narrated` prouve une règle avec une comparaison et une exigence transformée en implication. `gg_rule_governance` prouve qu'une règle qui ne gouverne rien est signalée. `wisecoding_narrated`, sa troisième scène, prouve un refus gouverné et mis en point de contrôle. `governance_narrated` prouve que pouvoir n'est pas devoir : une permission détenue sans l'autorité est un refus. `governance_lineage_narrated` prouve la décision gardée avec son autorité du moment. `01_platform_narrated` prouve la jointure des capacités : la caméra accordée, les paiements refusés, une capacité jamais demandée refusée avec une raison honnête. La garde n'utilise pas elle-même la capacité pour le hors ligne ; le bloc ci-dessus le fait, avec le niveau que la plateforme déclare pour elle. Le refus d'un type de règle inconnu, d'un fichier manquant et d'un nom en double repose sur les scènes 4 et 5 de `rulz_roundtrip_narrated`. C'est un test narré de la bibliothèque, et ce livre ne l'exécute pas comme l'une de ses gardes.

La section suivante met ce que fait le terrain face à ce que la bibliothèque a.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`graphruleset_narrated` 25 assertions
base/test/graph/graphruleset_narrated.ring

`gg_rule_governance` 29 assertions
base/test/graph/gg_rule_governance.ring

`wisecoding_narrated` 13 assertions
base/test/conversation/wisecoding_narrated.ring

`governance_narrated` 19 assertions
base/test/governance/governance_narrated.ring

`governance_lineage_narrated` 37 assertions
base/test/governance/governance_lineage_narrated.ring

`01_platform_narrated` 31 assertions
base/test/platform/01_platform_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en règles métier pour des gens du métier. Les tables de décision sont une norme ; une personne édite une cellule, pas un programme. Ici le changement est une règle reconstruite dans le code. La forme de fichier qui porterait une règle comme un texte, le fichier `.stzrulz`, lit le nom d'une règle, son type, sa gravité, sa fonction, ses paramètres et son message. Elle ne lit pas les lignes `when` et `then` de son propre exemple.

Le terrain mène en hors ligne. Le logiciel local d'abord est une pratique avec des bases de données qui se synchronisent et une décennie d'expérience. La bibliothèque a une permission.

Le terrain mène en gens. Une table de décision est une chose qu'un analyste formé sait déjà lire. Et la spécification propre de l'organisation, qui préférait la suite de l'éditeur, avait de bonnes raisons : ses utilisateurs y ont déjà des comptes.

La section suivante liste ce qui est construit, ce qui est conçu et ce qui est absent.

7. LA LIMITE

CONSTRUIT	une règle construite comme un objet sur un modèle, son nombre changé et le modèle jugé de nouveau ; le gouverneur de règles qui nomme une règle qui ne gouverne rien ; un refus gardé comme un point de contrôle qu'une personne peut lire, avec son contexte ; l'autorité mise face au risque ; une décision gardée avec son autorité du moment ; le hors ligne comme une capacité qu'un monde doit demander
CONÇU	le hub et son étude ; un langage de règles qu'un programmeur local peut apprendre ; les promesses de l'étude : qu'une règle est vérifiée avant de s'appliquer et peut être défaite
ABSENT	le hub, construit ; un mécanisme pour le travail fait hors ligne, gardé et envoyé plus tard ; un historique des versions d'une règle ; une plateforme client sur le moteur

Ce qui est construit : une règle comme objet, son nombre changé et le modèle jugé de nouveau ; le gouverneur de règles qui nomme une règle morte ; le point de contrôle ; l'autorité face au risque ; la filiation ; le hors ligne comme capacité.

Ce qui est conçu : le hub et son étude, dont le calendrier est dans le futur ; un langage de règles qu'un programmeur local peut apprendre ; et les promesses de l'étude sur les règles. L'exemple du langage dans l'étude est écrit dans une notation qui n'est pas celle de la bibliothèque. L'étude promet qu'une règle est vérifiée avant de s'appliquer, que son effet est montré avant d'être appliqué, qu'elle peut être défaite et que son historique est gardé. La bibliothèque n'a que les pièces. Ce sont le

gouverneur de règles, montré plus haut, et les refus d'un type de règle inconnu, d'un fichier manquant et d'un nom en double, qu'un test narré de la bibliothèque prouve et que ce chapitre ne montre pas.

Ce qui est absent : un historique des versions d'une règle avec un retour en arrière. La filiation enregistre la décision, et la règle est reliée de nouveau. Je n'ai trouvé aucun magasin de versions de règles, et ma recherche n'était pas exhaustive. Le hub, construit. Le hors ligne comme mécanisme : aucune classe de la bibliothèque ne garde des entrées sur un appareil et ne les envoie quand le réseau revient. Et une plateforme client sur le moteur Softanza.

Six portes suivent, une pour chaque lecteur.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Une règle est un objet avec une comparaison et une exigence, attaché à un modèle, et un gouverneur de règles vous dit quand elle ne gouverne rien.
L'analyste	L'étude promet que vous changez les valeurs d'une règle, pas son programme. Dans la bibliothèque, quand les valeurs changent, le modèle est jugé de nouveau avec la nouvelle portée.
Le concepteur	Un refus garde son contexte pour la personne qui le lira.
Le décideur	L'autorité n'est pas la permission : demandez quel niveau couvre quel risque, et lisez la filiation d'une décision.
Le formateur	La règle d'une organisation peut s'enseigner comme une phrase avec un nombre dedans.
L'agent	Vous préparez. Une personne décide. Et vous serez refusé, avec le niveau et le niveau de risque nommés.

9. EXERCICE

Choisissez une règle d'une organisation que vous connaissez qui contient un nombre, un plafond, un délai, un nombre d'approbations. Écrivez-la comme une règle avec une condition et une exigence, sur quatre cas inventés, deux qui la

satisfont et deux qui ne la satisfont pas. Avant de lancer quoi que ce soit, écrivez le rapport que vous attendez. Puis faites exprès une faute de frappe sur une propriété de la condition, et dites ce que le modèle rapporte, et ce que le gouverneur rapporte.

Lancez-la. La différence entre ces deux réponses est le chapitre.

Je vous laisse celui-ci. Le bloc ci-dessous est le deuxième bloc de la section 3 suivi des dernières lignes du bloc du gouverneur, avec un seul changement : la règle n'a pas de ligne `SetMessageQ`, donc elle ne porte aucun message.

HARO

```
oG.ClearAttachedRules()
oRule = new stzGraphRule("second-approval")
oRule.SetDomainQ("advances")
oRule.WhenQ("amount", "greaterthan", 200)
oRule.ThenQ("approvals", "greaterthan", 1)
oRule.ThenViolationQ("two approvals needed")
oG.AddRule(oRule)
oG.RulesReport()
oWatch2 = StzRuleGovernance("hub")
oWatch2.AddRule(oRule)
oWatch2.AddCase("advances", oG)
? len(oWatch2.CheckRules())
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez-le sur le modèle de la section 3. Comparez ce que le modèle rapporte avec ce qu'il rapportait quand la règle avait son message, et lisez ce que le gouverneur renvoie. Sous « J'ai refusé une règle silencieuse », la section 4 dit ce que le gouverneur fait d'une telle règle, et pourquoi. Je ne donne pas la réponse ici. Quand vous l'aurez résolu, vous saurez pourquoi le gouverneur signale une règle qui n'a pas de message.

Le chapitre suivant dit ce que les projets montrent, et la suite. Sa mesure est celle-ci : un projet montre exactement ce qu'un document, un fichier, une exécution et une personne nommée atteignent, et la page dit où ils s'arrêtent.

OÙ SONT LES PREUVES

NARRATION	stz-rules-as-graph-rules-narration.md
NARRATION	stz-governance-lineage-narration.md
SUR LE SITE	forged
SUR LE SITE	customers
DÉCISIONS DE L'ÉTUDE	12.13, 12.29

PARTIE VI · CHAPITRE 31

Ce que les projets montrent, et la suite

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 5 bloc(s) sur 5 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je dis, projet par projet, ce que les trois projets des chapitres 28 à 30 montrent aujourd'hui, et ce que chacun fait de l'étape suivante. Je commence par les trois habitudes du terrain pour dire ce qu'un travail a montré. Puis je lance cinq petits blocs où la bibliothèque montre, en refusant, où une étape s'arrête, et je dresse les tableaux : pour chaque projet, l'affirmation, le document ou le fichier qui l'appuie, et la suite. Je dis ce que j'ai refusé, ma mémoire comme preuve d'abord, et je donne le tribunal, le terrain et la limite dans les mots d'étape. Le principe est la phrase propre du chapitre : un projet montre exactement ce qu'un document, un fichier, une exécution et une personne nommée atteignent, et la page dit où ils s'arrêtent.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Le terrain a trois habitudes pour dire ce qu'un travail a montré.

La première est la référence : un logo sur une page d'accueil, une histoire de client de deux pages, la phrase « ils nous font confiance ». Elle montre que quelqu'un a payé ou signé. Elle ne dit pas ce qu'il a fait tourner, pendant combien de temps, ni ce qu'il en a dit.

La deuxième est l'échelle de maturité. Les niveaux de maturité technologique de la NASA, devenus ensuite une norme internationale, nomment la distance qu'une technologie a parcourue du laboratoire jusqu'au lieu pour lequel elle a été construite.

Le plus haut niveau est un système éprouvé en exploitation. C'est un vocabulaire d'étape, et ce livre en utilise un qui lui est propre (construit, conçu, absent).

La troisième est la section des limites. Une étude empirique doit énoncer les menaces à sa validité. Un modèle d'apprentissage automatique est livré avec une fiche dont la dernière partie est ses réserves. Un artefact soumis à une conférence de recherche reçoit un badge selon ce que quelqu'un d'autre en a fait. C'est la forme la plus proche de ce chapitre, et celle qu'on saute le plus souvent quand le lecteur est un acheteur.

Ce que les trois ont en commun, c'est que l'affirmation et sa preuve voyagent séparément. Une page dit « utilisé par », et la preuve est une confiance accordée à la page. C'est une connaissance générale, pas une étude du dossier.

2. LE PREMIER PRINCIPE

Le dossier donne deux phrases, et elles n'en font qu'une. Le site dit : « Rien n'est une référence tant qu'un document ne le prouve pas. » Et, de ce dont je me souviens : ce qu'« aucun fichier ne prouve n'est pas listé ».

Un projet montre exactement ce qu'un document, un fichier, une exécution et une personne nommée atteignent, et la page dit où ils s'arrêtent. C'est la phrase propre du chapitre, tirée des deux phrases du site. Les mots du site sont les deux cités plus haut, de sa page des clients et de sa page forgée.

3. LA FORME SOFTANZA

Ce chapitre est l'autre moitié des chapitres 28 à 30. Ceux-là disaient ce que trois projets ont appris à la bibliothèque. Celui-ci dit, projet par projet, ce que chacun montre aujourd'hui et ce qu'il fait de l'étape suivante. Des parties de Softanza sont en production, et d'autres sont en développement. Je n'ai pas écrit ce chapitre après coup, par modestie : la page forgée du site porte déjà une section sur le même sujet.

La forme de la réponse est un tableau de trois colonnes pour chaque projet : l'affirmation, le document ou le fichier qui l'appuie, et la suite. Je laisse une cellule vide quand je ne peux pas nommer ce qui la remplit. Et je ne remplirai pas la troisième colonne de mémoire.

Avant les tableaux, cinq petits blocs. Chacun est un lieu où la bibliothèque montre, en refusant, où une étape s'arrête, au lieu de passer en silence. Ils sont la phrase du chapitre, écrite comme du code.

Une dépendance déclarée et jamais liée est une erreur. La surface d'une solution est déclarée avant que rien ne soit lié. Un service déclaré que rien ne soutient est compté comme un constat, pas comme un passage, et le demander lève une erreur là où est la faute. C'est la première étape vers laquelle la leçon du restaurant pointe : lier un vrai service de paiement, ce que le dossier ne tient pas encore. Que répond le registre quand on lui demande le service de paiement et que rien ne le soutient ? Lancez-le :

HARO

```
oReg = new stzServiceRegistry("shop")
oReg {
        DeclareMany([ :mail, :payments ])
        ? @@( UnboundServices() )
        #--> [ "mail", "payments" ]
        ? Findings()[1][:invariant]
        #--> unbound-service
        ? IsSound()
        #--> 0
}
try
        oReg.Service(:payments)
catch
        ? cCatchError
done
#--> stzServiceRegistry(shop): nothing is bound for service 'payments'. Bind it (or BindLive it)
before use.
```

✓ exécuté : a affiché ce que ce bloc promet

Deux services déclarés, aucun lié, et le registre le dit lui-même : `nothing is bound for service 'payments'!`

Ce qu'une répétition peut affirmer, et ce que seule la cible peut affirmer. Le modèle de plateforme donne trois verdicts pour la même opération. *Refusé* : la cible l'interdit. *Natif* : l'hôte lui-même peut le faire. *Répété* : la cible le permet et l'hôte ne l'a pas, de sorte que l'exécution ne montre que le plan. Lire une broche sur

une machine de développement est répété, pas natif. Le mot dépend de la machine sur laquelle ce livre a été exécuté. Un PC Windows n'a pas une telle broche, et une carte qui en a une imprimerait `native` pour la même ligne. Que répond la portée de la carte à une lecture de broche, et à une demande de lancer un processus ? Lancez-le :

HARO

```
oSol = new stzPlatformProfile("kiosk")
oSol {
        DevelopedOn(:Windows)
        WithServer(:backend, :LinuxServer)
        WithApp(:firmware, :ESP32)
}
oFw = oSol.App(:firmware).System()
? oFw.ReadPin(4)
#--> rehearsed
try
        oFw.Spawn("worker")
catch
        ? cCatchError
done
#--> scope 'esp32' (espidf) forbids 'process' -- spawn 'worker'. This target has no such
capability.
oFw.Show()
#--> Scope: esp32 (espidf)
#      [rehearsed] read pin 4
```

✓ exécuté : a affiché ce que ce bloc promet

ce bloc touche the system ; la sortie montrée a été enregistrée le 2026-10-15 12:58 UTC+01:00

La lecture de la broche répond `rehearsed`, et la portée refuse le processus avec `forbids 'process'` !

Le commentaire de la bibliothèque, dans la classe qui abaisse une répétition en artefact réel, nomme l'étape suivante : la dernière, flasher une carte, « remains a PLANNED external action ».

L'adaptateur de paiement réel existe, et refuse de deviner. Le code du système de paiement réel existe. Son exécution contre le bac à sable de l'organisation n'a pas encore eu lieu, et c'est l'étape suivante. Sans identifiants, il refuse avant tout

appel réseau et dit ce qui manque. Que dit-il quand on lui demande des comptes et qu'il n'a aucun identifiant ? L'hôte de ce bloc n'existe pas, et rien ne l'atteint :

HARO

```
oAd = new stzPispiHttpAdapter()
oAd {
        WithParticipant("shop")
        WithBaseUrl("https://pispi.example")
        AsLive()
        ? IsSandbox()
        #--> 0
}
try
        oAd.Request("GET", "/comptes", [], [])
catch
        ? cCatchError
done
#--> stzPispiHttpAdapter: no credentials. Give it the store and an actor: WithSecretsFrom(oStore,
oActor).
? oAd.TokensFetched()
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

L'adaptateur répond `no credentials`, nomme ce qui comblerait le manque, `WithSecretsFrom(oStore, oActor)`, et `TokensFetched()` reste à `0`.

La bibliothèque a un script qui tente l'exécution réelle. Je l'ai lancé une fois, sans aucune de ses variables définie, pour ce chapitre. Il a imprimé le mot UNPERCEIVED, nommé les quatre variables qui manquent, et dit que rien de l'adaptateur réel n'est perçu tant qu'une personne nommée n'a pas vu un paiement arriver. Il s'est terminé sans échec. L'exécution a été reportée, par moi, pour une raison qui n'est pas celle de la bibliothèque : elle exige l'implication de l'organisation. Une exécution qui n'a pas pu avoir lieu est rapportée comme non perçue. Elle n'est rapportée ni comme un passage ni comme un échec.

Une fonction qui attend une personne nommée. Dans le volet du son, la bibliothèque tient comme donnée le verdict d'une personne sur une chose que seule une personne peut juger. Tant que le verdict est UNPERCEIVED, le verbe qui

revendiquerait le résultat refuse et dit pourquoi. Là où une personne a dit oui, le verdict porte la date et ses mots. Que répond la voix à formants quand on lui demande de chanter, et que contient le verdict de la voix réaccordée ? Lancez-le :

HARO

```
oVoice = StzSoundFormantVoiceQ()
oVoice {
        Sing("la la la", "c4 e4 g4")
        ? Refusals()
        #--> 1
        ? LastError()
        #--> Sing is not available YET: the formant voice has not been heard by the author
(sound_mu5_demo.ring plays it). SAPI was measured and does not hold a pitch (measured 2026-09-26:
+-6 st asked, +-2.5 st given, up to 407 cents off, 286-575 cents of wander within a syllable --
it speaks, it cannot hold a note). Until the author calls the formant voice singing, Vowel and
VowelsQ are what this voice honestly does
}
aVerdicts = StzSoundSingingVerdict()
? aVerdicts[:formant][:verdict]
#--> UNPERCEIVED
? aVerdicts[:retuned][:verdict]
#--> SINGING
? aVerdicts[:retuned][:date]
#--> 2026-09-27
? aVerdicts[:retuned][:said]
#--> the retuned voice is somehow singing, open Sing()
```

✓ exécuté : a affiché ce que ce bloc promet

La voix à formants a refusé de chanter : `Refusals()` vaut `1`, et le message commence par `Sing is not available YET` ! Le verdict de la voix à formants reste `UNPERCEIVED`. Le verdict de la voix réaccordée se lit `SINGING`, avec la date `2026-09-27` et les mots `the retuned voice is somehow singing, open Sing()`. C'est la forme de la ligne que le chapitre 33 fixe comme prochaine étape pour l'exécution d'un client : une date, une personne, et une phrase. Le dossier porte aussi le nom de la personne à côté, et je ne l'imprime pas ici. SAPI, dans le message, est l'interface vocale du système d'exploitation.

Un contrôle préalable qui dit « faisable » sans toucher un hôte. Le contrôle préalable du déploiement refuse un hôte trop petit. Pour un hôte dont le fournisseur est un cloud, il dit faisable et « provision on aws », sans aucun compte. Une

faisabilité est un plan, pas une vérification. Le fichier de conception nomme l'étape suivante clairement : la vérification contre un vrai hôte ou compte, qui attend l'infrastructure. Les chiffres ci-dessous sont illustratifs. Que répond le contrôle préalable à un hôte trop petit, et à un fournisseur cloud sans aucun compte ? Lancez le bloc :

HARO

```
oBrain = new stzDelivery("shop")
oBrain.AddBackend(:api, :LinuxServer)
oBrain.RequiresIn(:api, StzResourcesQ().SetMemoryQ(2048).SetComputeQ(2).SetStorageQ(20))
oSmall = new stzDeploymentSite("tiny")
oSmall.SetKindQ(:Server).SetCapacityQ(StzResourcesQ().SetMemoryQ(512).SetComputeQ(1).SetStorageQ(
5))
oTiny = new stzDeployment(oBrain)
oTiny.SetTarget(:api, oSmall)
? oTiny.Feasible()
#--> 0
? oTiny.Feasibility()[1][5]
#--> SHORTFALL -- host too small
oCloud = new stzDeploymentSite("pool")
oCloud.SetKindQ(:Server).SetProviderQ(:aws)
oPool = new stzDeployment(oBrain)
oPool.SetTarget(:api, oCloud)
? oPool.Feasible()
#--> 1
? oPool.Feasibility()[1][5]
#--> provision on aws
```

✓ exécuté : a affiché ce que ce bloc promet

Le petit hôte reçoit `SHORTFALL -- host too small`, et le cloud sans compte est faisable, avec `provision on aws` !

Voici maintenant les tableaux. Chacun s'ouvre sur ce que le projet a livré ou fait tourner, et se termine sur la suite. Chaque cellule vient du dossier, lu le 8 octobre 2026. Un compte de lignes ou une date de fichier que j'ai lus moi-même sont étiquetés comme miens.

LE RESTAURANT À LYON

AFFIRMATION	PREUVE	LA SUITE
La plateforme existe, et sa première livraison a été faite	le site : un avenant clos le 4 août 2026 ; le commit « iteration 1 » du projet du 15 août ; un serveur de 565 lignes que j'ai lu ; le dernier commit, 25 août 2026, 13 h 07	le 25 août 2026 est le dernier enregistrement daté ; le prochain est celui que l'étape suivante écrira : la production, une vraie commande, la barrière de mise en service prévue
La première itération a été livrée, et une journée de terrain a suivi	le site : « première itération livrée » ; l'article du projet : une journée de mesures de terrain avec deux téléphones, le 19 août	la phrase du propriétaire sur la manière dont il s'en sert est le prochain enregistrement à recueillir ; les mots du dossier sont « première itération livrée » (le site) et « une journée d'essais sur le terrain » (l'article du projet)
Aucune croissance du périmètre pendant un cycle (le Verrou, chapitre 28)	le pacte et le carnet du Verrou : des documents privés	le Verrou comme code ; le chapitre 28 montre ce que la bibliothèque donne de plus proche
On peut essayer des paiements sans abonnement, et un faux ne part jamais	le registre et la barrière de production, avec leurs gardes	l'adaptateur de paiement réel pour le restaurant, « a later desk » dans le propre fichier de conception de la bibliothèque ; dans la garde du registre de la bibliothèque, un bac à sable est lié, et la liaison réelle est l'étape qui suit ; la solution propre du restaurant n'appelle pas le registre
La constellation de mondes	la carte des mondes du restaurant ; la classe de super-application de la bibliothèque, avec une garde	le restaurant sur la constellation : ses mondes sont aujourd'hui des pages web en fichier unique, et la carte dit qu'elle est une carte, pas une réalisation
Un budget de vitesse est une promesse	l'objet de niveau de service de la bibliothèque, avec une garde	une mesure sur un vrai réseau 4G : le tableau de bord du projet au 23 août la reporte au cycle suivant
Les règles d'interface ont été forgées avec le propriétaire, le téléphone en main	le fichier du projet dit onze itérations avec l'utilisateur réel	qui tenait le téléphone : le dossier ne le tient pas, et mon étude du dossier a confondu les deux faits

LA BANQUE À NIAMEY

AFFIRMATION	PREUVE	LA SUITE
La banque a mon produit d'évaluation, installé sur son propre réseau interne sous licence	le guide d'installation remis à la banque ; un fichier de licence	le récit de la banque elle-même : qui déclare l'organigramme, depuis combien de temps, combien de personnes l'utilisent. Et une date de livraison : les fichiers portent des horodatages du système de fichiers, qui ne sont pas des dates
Le produit est construit avec Softanza	j'ai construit les deux	le produit qui appelle la bibliothèque. Aujourd'hui, c'est une application Python avec un petit serveur web ; j'ai cherché dans son code, et je n'y ai trouvé aucun contrôle régional ou de Bâle et aucun appel à la bibliothèque
Les règles du régulateur jugent l'organigramme	le validateur de trois règles de la bibliothèque et celui de la séparation des tâches, avec leurs gardes	les validateurs à l'intérieur du produit ; un article du texte de la banque centrale pour les trois codes de règle ; un constat lu par un auditeur
Un fichier de règles pour la banque	un fichier de deux règles régionales qui se charge	son premier jugement : sur un organigramme sans conseil, il rapporte sain aujourd'hui
Les règles d'interface du produit viennent de la banque	cinq règles de la constitution de l'interface	la ligne de preuve de la première d'entre elles dit « A banking supervision product (April 2026) » et ne nomme aucune banque. Mon étude du dossier les attache à cette banque, et le dossier non

L'ORGANISATION SUR LE TERRAIN

AFFIRMATION	PREUVE	LA SUITE
L'organisation a choisi mon offre	l'offre du 5 août 2026 a été retenue ; l'étude de conception, construite avec ses équipes du 22 septembre au 8 octobre 2026	le hub de l'organisation, pas encore construit ; la version 1.2 de l'étude est soumise à validation. Et qui a choisi : le dossier est muet
Les règles sont des données qu'une personne change sans programmeur	les clauses de règles de la bibliothèque et le gouverneur qui nomme une règle morte	une forme texte pour les règles à clauses ; un historique des versions avec retour en arrière, que l'étude de l'organisation promet ; sa propre notation de règles n'est pas celle de la bibliothèque

AFFIRMATION	PREUVE	LA SUITE
Un agent d'IA prépare, une personne décide, avec l'autorité du moment	l'objet de gouvernance et sa filiation, avec leurs gardes	une personne liée à un compte connecté ; un hub où le faire
Hors ligne par défaut	le mot de l'étude de l'organisation selon lequel une boîte d'envoi côté navigateur est déjà en service ; c'est un paquet frère extérieur à la bibliothèque, que le chapitre 30 nomme et que je n'ai pas exécuté	le mécanisme dans la bibliothèque est à venir : ni file, ni boîte d'envoi, ni synchronisation pour l'instant. Le site l'appelle « un manque, pas une leçon apprise ». Le dernier bloc du chapitre 30 en est le tout

SUR LES TROIS

AFFIRMATION	PREUVE	LA SUITE
Les plateformes tournent aujourd'hui sur des piles ordinaires	la page propre du site : des mondes web en fichier unique, une application Python, le hub de l'organisation non construit ; et un petit serveur Node que j'ai lu	la première plateforme sur le moteur. Le site le dit, et chaque dépôt que j'ai ouvert est d'accord
Leurs déclarations « sont en cours d'écriture comme celles de Softanza »	la page des plateformes et la page forgée, comme intention	les déclarations en fichiers sont le prochain morceau à écrire. J'ai cherché, le 8 octobre 2026, et je ne les ai pas trouvées comme fichier : dans le dépôt du restaurant, aucun fichier de code ne déclare une application, une constellation ou une plateforme ; dans la livraison de la banque, aucun ; dans le dossier d'étude de l'organisation, aucun (son annexe utilise une notation qui n'est pas celle de la bibliothèque). Le dossier montre l'intention et les cartes ; les déclarations sont l'étape qui suit
Un changement de moteur,	le principe : la réécriture a été refusée	le déplacement lui-même ; le dossier ne dit pas encore qui se déplace en premier ni qui le perçoit

AFFIRMATION	PREUVE	LA SUITE
pas de plan		
Une solution écrit sa fiche de garanties en cinq lignes	la vision la demande	la première fiche est à venir : aucune solution n'en a encore une, et aucune classe, fixture ou garde de la bibliothèque ne tient la fiche
Le runtime en dessous est celui du langage lui-même	la charte ratifiée le 1er octobre 2026 ; ses étapes passent sur ses propres fixtures	Haro qui fait tourner la bibliothèque est l'étape suivante. Une note de session du 5 octobre 2026 rapporte que le runtime de pont passe chaque assertion que l'ancien runtime passe sur la bibliothèque, mais que tous les programmes ne s'accordent pas sur chaque ligne stable : sur 5 112 programmes, 3 425 sont identiques sur chaque ligne stable, 1 267 de plus ne diffèrent que par la ligne qui imprime le temps écoulé, et 19 diffèrent en nature, tel que le chapitre 27 lit la note ; le passage sous Linux est l'étape d'après. Je n'imprime aucun pourcentage. Les blocs de ce livre s'exécutent sur l'ancien runtime, 1.27
Un hôte réel vérifie un déploiement	la conception du déploiement	la vérification attend l'infrastructure et est l'étape suivante. Les sites locaux tournent pour de vrai ; le transfert distant est « a later slice »
Le volet des paiements fonctionne contre le bac à sable de l'organisation	le jumeau, servi par HTTP, avec une garde	l'exécution du bac à sable : reportée, non perçue jusqu'ici. La production ne vient jamais de ce volet
Une personne nommée a dit ce qu'elle a vu	un tambour, et la voix réaccordée, dans un autre volet	la même ligne pour n'importe laquelle des trois plateformes, qui est la prochaine étape

CE QUI RESTE À MESURER

Mon étude du dossier l'a lu et ne l'a pas exécuté, et elle le dit ; voici ce qu'elle laisse à mesurer ensuite. Savoir si le runtime de Haro fait tourner la bibliothèque est la première mesure à venir. Savoir si deux mesures de vitesse publiées tiennent encore aujourd'hui est la deuxième, et les deux ont été faites sur l'ancien runtime. Certaines gardes du dossier système ont pris leurs totaux dans des documents de conception, et l'un de ces documents dit encore quarante-cinq là où sa garde en tient maintenant davantage ; le document est le prochain à mettre à jour. Les quarante et une images de la première partie de la galerie ont été jugées par un modèle d'IA, et qu'une personne les regarde est l'étape suivante. Le budget de vitesse du restaurant attend sa mesure sur un vrai réseau 4G. Chacune de ces mesures est une mesure pas encore prise, non un échec.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé ma mémoire comme preuve. Le site ne liste rien dont je me souvienne et qu'aucun fichier ne prouve. La même règle liait la banque : je connais le produit, parce que je l'ai construit, et ce que les gens de la banque en font n'est pas encore dans un fichier.

J'ai refusé d'appeler la bibliothèque l'origine de ce qu'elle n'utilise que comme exemple. La page forgée dit « exemple » là où la bibliothèque tient un projet comme son exemple et non comme son origine. Et je ne répéterai pas une phrase de mon étude du dossier qui dit qu'une leçon est « née du besoin du client ». Le registre et le SLA ont été commités pour la première fois après le renversement du 4 juillet, selon le journal de la bibliothèque tel que le lit le chapitre 28. L'ordre des commits ne soutient pas « née de ». Ce que je peux dire, c'est « avait besoin » et « porte ».

Je garde les mots du dossier pour ce que chaque client fait tourner. Le dossier montre, pour la banque, une application Python installée sous licence et construite à côté de la bibliothèque par moi, et, pour le restaurant, une première livraison et une journée de terrain, le dossier s'arrêtant au 25 août. Les mots que le dossier appuie sont *construit par moi, sous la discipline que la bibliothèque porte désormais*, pour la banque, et, pour le restaurant, *première itération livrée* (le site) et *une journée d'essais sur le terrain* (l'article du projet). Le chapitre 1 le dit avec ses propres mots, et ce paragraphe est l'endroit où se tient la formulation plus exacte du dossier.

J'ai refusé un mot d'étape que le dépôt ne porte pas. La page du domaine de projets du site le dit : ne jamais appeler disponible avant que ce ne soit. Un travail qui atteint une personne par un sens est perçu par une personne. Jusque-là il est non perçu, ce qui est un état et non un échec.

J'ai refusé d'imprimer un pourcentage. J'ai dit à voix haute jusqu'où le runtime de pont est arrivé, comme une part du chemin, et ce chiffre ne s'imprime pas sans la garde qui le produit. Le facteur de la section 6 est un autre nombre : un rapport de vitesse mesuré au chapitre 27, ligne par ligne, contre un pair rapide.

5. LE TRIBUNAL

`service_registry_narrated` prouve qu'un service déclaré que personne ne soutient est un constat d'erreur, qu'un service non lié lève une erreur, et qu'un bac à sable en production est une violation. `full_scope_model_narrated`, scènes 5 à 8, prouve les trois verdicts : refusé, répété, natif. `payments_live_adapter_narrated` prouve l'adaptateur contre un jumeau servi par HTTP, un adaptateur sans identifiants qui refuse de deviner, et le scénario qui dit que l'étape (b) est non perçue. Elle démarre de vrais processus serveur, et le livre ne l'exécute pas ; il ne fait que compter ses assertions. `sound_mu5_narrated` prouve, dans sa cinquième scène, que tant que le verdict est non perçu `Sing` ne retourne rien et dit pourquoi. `deployment_narrated` prouve qu'un hôte sous-dimensionné n'est pas faisable, et qu'un assistant répète et ne valide rien ; elle écrit un dossier temporaire à côté d'elle, donc le livre ne l'exécute pas non plus. `orgrule_narrated` prouve les règles universelles de l'organigramme et la séparation des tâches. `14_bceao_governance`, une assertion, exécute les règles régionales sur un organigramme sans conseil et sans fonction de gestion des risques, et deux des trois se déclenchent, comme le dit le chapitre 29. `governance_narrated` prouve l'autorité face au risque, avec sa filiation.

Le tribunal du sujet de ce chapitre n'est pas une garde. C'est une ligne du dossier, de la forme que prescrit la loi du domaine de projets : une date, une personne, et ce qu'elle a dit. La bibliothèque tient une telle ligne dans ses données, dans le volet du son. Le domaine de projets en tient une dans son dossier, et le chapitre 33 la raconte.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`service_registry_narrated` 93 assertions
base/test/system/service_registry_narrated.ring

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

`payments_live_adapter_narrated` 125 assertions
base/test/system/payments_live_adapter_narrated.ring

`sound_mu5_narrated` 20 assertions
base/test/sound/sound_mu5_narrated.ring

`deployment_narrated` 63 assertions
base/test/system/deployment_narrated.ring

`orgrule_narrated` 22 assertions base/test/graph/orgrule_narrated.ring

`14_bceao_governance` 1 assertions
base/test/orgchart/14_bceao_governance.ring

`governance_narrated` 19 assertions
base/test/governance/governance_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en production. Des piles ordinaires font tourner de vraies charges pour de vrais utilisateurs. C'est une connaissance générale, pas une étude du dossier. La page propre du site place aujourd'hui les plateformes de cette partie sur des piles

ordinaires, le hub n'étant pas construit ; déplacer leurs déclarations sur le moteur propre de Softanza est l'étape suivante.

Le terrain mène en références à grande échelle. Le site imprime deux références et une étude de conception, et ne cite aucun nombre de pays et aucun nombre de programmeurs. Un acheteur qui lit un mur de logos en voit davantage.

Le terrain mène en infrastructure réelle. Les outils de provisionnement pilotent de vrais fournisseurs chaque jour, ce qui est une connaissance générale. Le déploiement de la bibliothèque atteint aujourd'hui des sites locaux pour de vrai, et le vrai hôte est l'étape suivante ; le cinquième bloc montre où passe la ligne.

Le terrain mène en matériel sur le terrain. La machine déclarée démarre et se juge elle-même dans un émulateur, et la vision note qu'aucun client n'est encore dessus. Son texte de garanties dit quatre promises sur quatre, trois tenues dans l'émulateur, et ne porte aucune décision d'emplacement ; la vision ajoute que la quatrième, survivre à une coupure, attend le chien de garde sur la carte. La carte est commandée.

Le terrain mène en vitesse. Le runtime contre lequel Softanza s'est mesurée était plus rapide sur chaque ligne, d'un facteur entre un et deux : de 1,04 à 1,88 fois, les six lignes que le chapitre 27 imprime d'après le mémo du 13 septembre 2026. Ce facteur est une mesure, non le pourcentage d'avancement que la section 4 refuse. Les deux mesures ont été faites sur l'ancien runtime ; la vitesse propre de Haro est la mesure encore à venir.

Le terrain mène en images jugées par des personnes. Quarante et une images et sections de texte de la première partie de la galerie ont été jugées par un modèle d'IA : vingt justes, neuf fausses, douze incertaines. Le dossier dit qu'aucune personne ne les a encore regardées, et ce regard est l'étape suivante.

Là où je pense que le dossier se distingue, c'est que sa limite est une section d'une page, avec un titre. Chaque étape a un mot. Une fonction peut être construite et refuser encore jusqu'à ce qu'une personne nommée ait parlé. Et une exécution réelle qui n'a pas pu avoir lieu est rapportée non perçue, jamais comme un échec et jamais comme un passage.

7. LA LIMITE

CONSTRUIT	ce que les trois projets ont livré, chacun avec un document (une première livraison suivie d'une journée de terrain ; un produit installé sous licence ; une étude de conception construite avec les équipes de l'organisation) ; les trois leçons que la bibliothèque porte, chacune avec une garde (un faux ne part jamais ; un organigramme jugé selon les règles d'un régulateur ; l'autorité face au risque, avec sa filiation) ; des refus qui montrent une absence au lieu de la cacher
CONÇU	la fiche de garanties en cinq lignes ; un runtime qui est celui du langage lui-même et fait tourner la bibliothèque ; la machine déclarée sur une carte
ABSENT	toute plateforme client sur le moteur ; les déclarations des clients écrites comme celles de Softanza elle-même, en fichiers ; un mot de la banque ; le dossier du restaurant après le 25 août 2026 ; le hub de l'organisation ; un mécanisme hors ligne dans la bibliothèque ; une exécution réelle des paiements contre le bac à sable de l'organisation ; le verdict d'une personne nommée sur tout cela

Ce chapitre tout entier est un état d'étape. En voici la somme dans les mots d'étape.

Ce qui est construit : ce que les trois projets ont livré, chacun avec un document (une première livraison au restaurant, suivie d'une journée de terrain ; un produit installé à la banque sous licence ; une étude de conception construite avec les équipes de l'organisation), et les trois leçons qu'ils ont données à la bibliothèque. *Un faux ne part jamais* : le registre et la barrière de production. *Une organisation est jugée par son propre organigramme selon les règles d'un régulateur* : l'organigramme et ses validateurs. *L'autorité n'est pas la permission, et une décision garde son autorité du moment* : l'objet de gouvernance. Chacune a un fichier de la bibliothèque qui la porte, une garde qui se résout, et un bloc déjà gardé dans un chapitre voisin. L'étape suivante de chacune est dans les tableaux. Dans la garde du registre de la bibliothèque, un bac à sable est lié, et l'adaptateur de paiement réel est l'étape qui suit ; le produit de la banque n'appelle pas le validateur d'organigramme. Le hub de l'organisation n'est pas encore construit, et son mécanisme hors ligne est à venir.

Ce qui est conçu : la fiche de garanties en cinq lignes ; le runtime qui est celui du langage lui-même, faisant tourner la bibliothèque ; la machine déclarée sur une carte.

Ce qui est absent : toute plateforme client sur le moteur ; les déclarations des clients écrites comme celles de Softanza elle-même, en fichiers ; un mot de la banque ; le dossier du restaurant après le 25 août ; le hub de l'organisation ; un mécanisme hors ligne dans la bibliothèque ; une exécution réelle des paiements contre le bac à sable de l'organisation ; le verdict d'une personne nommée sur tout cela.

La somme de toutes les étapes suivantes est la ligne que le chapitre 33 fixe comme prochaine étape : *la solution a tourné sur le moteur propre de Softanza, et voici ce que l'utilisateur a dit*. Les déclarations des clients en sont le premier morceau, et elles sont absentes aujourd'hui. Le site dit qu'elles sont en cours d'écriture comme celles de Softanza elle-même. Le 8 octobre 2026, j'ai cherché un fichier de ces déclarations dans le dépôt du restaurant, dans la livraison de la banque et dans le dossier d'étude de l'organisation, et je n'en ai trouvé aucun. Le 9 octobre 2026, j'ai cherché de nouveau, en lecture seule, dans treize dépôts voisins de la bibliothèque, et je n'y ai trouvé aucun fichier qui déclare la plateforme d'un client comme celle de Softanza elle-même. Le mot du chapitre pour ces déclarations est donc *absent*, et la ligne d'étape en tête le dit.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Votre démonstration est une répétition jusqu'à ce qu'une cible l'ait exécutée. Demandez à la portée si la ligne était native, répétée ou refusée, et écrivez le même mot à côté de vos propres affirmations.
L'analyste	Quand vous citez un projet, écrivez trois colonnes : l'affirmation, le document, et la suite. Ne remplissez pas la troisième de mémoire.
Le concepteur	Une fonction qu'aucune personne n'a vue ni entendue part fermée, et le dit. L'étape est un état de la conception et non une excuse. Je tiens cela pour une vue, non pour un résultat. Ce qui la soutient sur cette page, c'est le volet du son, où `Sing` refuse tant qu'une personne n'a pas entendu la voix.
Le décideur	Lisez le mot d'étape avant le témoignage. « Construit » est une affirmation sur une garde. « Tourne chez un client » est une affirmation sur un document. Vous pouvez le demander par écrit, avec un nom à côté.
Le formateur	Enseignez la différence entre un projet qui a instruit la bibliothèque et un projet qui tourne dessus. Le premier est montré ici, et le second est la prochaine étape, au chapitre 33. Non perçu est un état, pas un échec.
L'agent	Vous pouvez lire les tableaux et citer n'importe quelle cellule. Vous ne pouvez pas transformer une cellule absente en cellule présente. Si vous le faites, le prochain lecteur du dossier trouvera votre phrase et le fichier manquant côte à côte.

9. EXERCICE

Déclarez la surface d'un de vos projets : un service de courrier, un service de paiements, et un de plus de votre choix. Avant de lancer quoi que ce soit, écrivez la ligne que vous attendez sous chacune de trois questions : quels services sont non liés ; si le registre est sain ; quels services sont non liés une fois qu'un bac à sable est lié au service de courrier.

HARO

```
oReg = new stzServiceRegistry("my-project")
oReg.DeclareMany([ :mail, :payments, :your_service ])
? @@( oReg.UnboundServices() )
#--> what you expect
? oReg.IsSound()
```

```
#--> what you expect
oReg.Bind(:mail, new stzMailSandbox())
? @@( oReg.UnboundServices() )
#--> what you expect
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Puis, sur papier, faites le tableau de trois colonnes pour le projet que vous connaissez le mieux : l'affirmation, le document ou le fichier qui l'appuie, et la suite. Laissez vide ce que vous ne pouvez pas nommer. Sur la dernière ligne, écrivez qui l'a perçu, à quelle date, et ce qu'il a dit, ou le mot « non perçu » et pourquoi.

L'exercice n'a pas de réponse à vérifier d'abord. La question que pose le tuteur est : quelle cellule avez-vous remplie de mémoire ?

Je vous laisse celui-ci, exprès. Ci-dessous, le premier bloc de ce chapitre avec une ligne de plus : elle lie le service de paiements au bac à sable de la bibliothèque avant que l'on demande au registre s'il est sain.

HARO

```
oReg = new stzServiceRegistry("shop")
oReg.DeclareMany([ :mail, :payments ])
oReg.Bind(:payments, new stzPaymentsSandbox())
? oReg.IsSound()
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez-le et lisez ce que répond `IsSound()`. Demandez-vous ensuite si un registre sain est un registre prêt à livrer, et ce que dit la garde de la section 5 d'un faux encore lié en production. Je ne donne pas la réponse : la section 5 nomme la garde qui la tient, et la section 7 dit ce que le projet a lié et ce qu'il n'a pas lié. Quand vous aurez résolu cela, vous saurez ce qu'un registre sain ne dit pas du service qui se tient derrière lui.

Le chapitre 32 se tourne d'abord vers la manière dont beaucoup d'esprits ont travaillé sur ce seul ouvrage, et vers la règle qu'ils gardent : chaque loi cite l'incident qui l'a gagnée, et une loi nouvelle sans un échec daté derrière elle est une préférence.

Le chapitre 33 referme ensuite le livre sur la prochaine étape, une solution qui tourne sur le moteur propre de Softanza.

OÙ SONT LES PREUVES

NARRATION	`stz-built-on-real-world-need.md`
NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stz-getting-paid-and-paying-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
SUR LE SITE	customers
SUR LE SITE	forged
SUR LE SITE	platforms
SUR LE SITE	estate
DÉCISIONS DE L'ÉTUDE	12.5, 12.10, 12.13, 12.21, 12.22, 12.29, 12.30

PARTIE VII

Ce qui vient

32 · Beaucoup d'esprits sur un seul ouvrage

33 · La ligne suivante : une solution sur son propre moteur

PARTIE VII · CHAPITRE 32 · COLONNE

Beaucoup d'esprits sur un seul ouvrage

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:00 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 3 bloc(s) sur 3 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je montre comment Softanza a été écrite depuis deux ans : par de nombreuses sessions d'un modèle d'IA sur une seule machine, sans cadre logiciel, avec seulement des fichiers, un historique de versions et quelques hooks. Je parcours les règles que le dossier a payées, un échec à la fois, et les quatorze règles de la maison de la bibliothèque qu'un agent d'IA peut lancer avant de commiter. Je dis ce que j'ai refusé : un cadre logiciel, un coordinateur qui note son législateur, l'automatisation au maximum, une décision prise en mon absence, un protocole vendu comme un produit. Et je dis la perte que j'ai acceptée : ma main s'est amincie là où les sessions ont écrit. Le principe est une phrase des sessions elles-mêmes : chaque loi cite l'incident qui l'a gagnée.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Quand le terrain parle de nombreux agents d'IA qui travaillent sur un seul corps de code, il pense en général à un cadre logiciel. Un programme contient plusieurs agents, un superviseur route les messages entre eux, et la coordination est une fonction de ce programme. L'AutoGen de Microsoft, CrewAI et le LangGraph de LangChain sont les noms que vous rencontrerez, tous apparus ces trois dernières années.

Quand le terrain parle des règles que ces agents suivent, il pense à un fichier que l'agent lit au démarrage, une page d'instructions dans le dépôt. Et quand il parle de nombreux auteurs sur un seul dépôt, il pense à ce que les équipes ont toujours employé : des branches, la relecture, des hooks qui s'exécutent avant un commit, une liste de qui possède quel dossier. Ce sont de bons outils, et je les utilise tous les jours.

Ce que j'ai fait à la place commence par une phrase.

2. LE PREMIER PRINCIPE

La phrase à laquelle cette partie du dossier revient sans cesse a été écrite par l'une des sessions, et je la garde parce qu'elle est juste : **chaque loi cite l'incident qui l'a gagnée. Une loi nouvelle sans un échec daté derrière elle est une préférence, et sa place est dans un guide de style.**

3. LA FORME SOFTANZA

Softanza n'a pas été écrite longtemps par une seule main. Depuis deux ans, la plupart de son nouveau code a été écrit par des sessions d'un modèle d'IA, plusieurs à la fois, sur une seule machine. Ce travail couvre une vingtaine de dépôts qui dépendent les uns des autres. Je tranche ; elles proposent, construisent, mesurent et rendent compte. Le dossier m'appelle le Principal. Dans ce chapitre, je dirai je.

Il n'y a pas de cadre logiciel ici. Pas de superviseur, pas de courtier de messages, pas de graphe d'agents. L'arrangement, ce sont des fichiers et un historique de versions. Un dépôt, le central, tient le protocole : comment une session rend compte, comment elle demande, comment elle clôt. Chaque dépôt a une boîte aux lettres. Chaque session écrit ses messages sous une forme fixe, huit clés dans un ordre fixe. Les huit clés : par qui, à propos de quoi, pourquoi, ce qui a été fait, l'état des choses nommées, ce qui attend et qui l'attend, ce qui vient ensuite, et une ligne de jugement. Chaque message est aussi ajouté à un fichier daté, pour que le raisonnement du jour ne soit jamais seulement dans une fenêtre de discussion qui se ferme.

Ce qui en fait une discipline et non un système de classement, c'est que chaque règle a été payée.

Indexer par chemin explicite. Plusieurs sessions écrivent dans le même dépôt en même temps. Deux fois, une session a ajouté d'un coup tous les fichiers modifiés à son commit, et a emporté le travail d'une autre session. Une fois elle a détruit net une modification, une fois elle a poussé la moitié du déplacement d'un collègue entre deux dépôts. La seconde fois, rien n'a été perdu, et le dossier dit que c'était de la chance, pas de la méthode. Alors maintenant un hook refuse la commande avant qu'elle ne s'exécute. Il a été testé sur vingt formes de la commande, et dix-huit se sont comportées comme écrit. Des deux autres, `git stage -A` a été réparée le jour même ; l'autre, que le hook refuse aussi un texte qui ne fait que citer la forme, est un coût accepté, rencontré à la section 6.

Lire l'heure sur l'horloge, ne jamais la composer. Une session écrit une heure sur chaque message. Les sessions, il s'avère, inventent des heures : elles écrivent une heure qui leur paraît juste après la précédente. Personne ne s'en aperçoit jusqu'à ce qu'une heure inventée passe minuit et qu'un message soit classé sous le mauvais jour. Comment la session centrale s'en est-elle sortie quand elle s'est mesurée elle-même ? Cinq de ses vingt-cinq horodatages avaient été composés ! Le pire était décalé de quarante-deux minutes, et quatre des cinq étaient plus anciens que la loi qui les interdisait. Sur les autres bureaux, trente-quatre horodatages sur cent cinquante et un étaient composés, sur treize des dix-sept bureaux mesurés, un bureau étant un dépôt avec sa boîte aux lettres. Les lignes derrière ce chiffre sont publiées. La première fois qu'il l'a été, les lignes avaient été jetées, et le chiffre n'a pas pu être défendu trois heures plus tard. Cinq semaines après la loi du 25 août, la session centrale a de nouveau composé un horodatage. Quatre avertissements écrits ne l'avaient pas arrêtée ; un pair qui lisait un fichier l'a fait. La note qu'elle a laissée est la meilleure phrase du protocole : un rappel n'est pas un mécanisme.

Les mots nomment des états, jamais des personnes. Une session a un jour rapporté que mon arbre de travail était « sale ». Un travail qui n'est pas encore commité n'est pas sale, et souvent il ne contient aucune erreur ; et un mot qui juge le travail juge celui qui l'a fait. Alors le dossier tient une table des mots que le domaine de projets n'emploie pas, avec le mot qu'il emploie à la place.

Ce qu'aucun compteur ne peut voir. Que disaient les compteurs du volet sonore de la bibliothèque ? Six cent vingt-deux assertions réussies sur dix-neuf gardes, et pourtant, pour tout auditeur, il était cassé ! J'ai entendu quatre défauts qu'aucun compteur n'avait signalés : une voix trop rapide pour être comprise, une phrase coupée avant sa fin. La loi qui en est sortie dit qu'une chose que personne n'a perçue est dans un état, pas dans l'échec : non perçu est un état, et la page doit le dire. C'est le même instinct que l'étape que porte chaque page de ce livre, Brouillon, jusqu'à ce que je l'aie approuvée.

La boucle se compte en secondes, pas en minutes. J'ai arrêté deux sessions sur la même exécution de test de huit minutes. Elle a été mesurée : six secondes sur les huit minutes étaient de la couverture, le reste était trois sections qui refaisaient le rendu du monde pour tester une seule propriété de pixel. Elle a été réécrite pour durer vingt secondes. J'avais écrit, de mes propres mots, qu'elle « prend beaucoup de temps et devrait être repensée ». La loi a été écrite par une session ; la phrase était la mienne.

Ne jamais demander sans dire comment. Une session m'a un jour demandé de vérifier si un chiffre était juste, sans me donner aucun moyen de le vérifier. J'ai répondu : s'il vous plaît, ne demandez jamais une chose sans me dire comment l'obtenir. Maintenant chaque demande adressée à une personne porte la commande exacte, le chemin exact, la ligne exacte à lire.

Vous voyez la forme. Un échec, daté. Une règle, écrite là où la session suivante la lira. Puis, quand la règle est enfreinte malgré tout, un mécanisme. Certaines lois sont arrivées à la dernière étape, d'autres sont encore une règle écrite, et le dossier dit lesquelles.

Là où les lois touchent le code, la bibliothèque elle-même peut les tenir. Ses quatorze règles de la maison sont des validateurs, cinq sur un graphe du code et les autres sur le texte : une loi du document de conception de la bibliothèque, rendue exécutable. Ce sont des règles qu'un agent peut lancer avant de commiter, et non du folklore qu'il doit deviner. L'une d'elles porte sur les mots. Que dit-elle d'une méthode nommée `KillOrder`, et le code est-il encore propre ? Exécutez-le :

```
HARO
```

```
cCode = "class Shop" + nl + "def KillOrder()" + nl + "  return" + nl
aFindings = StzCheckCode(cCode)
? aFindings[1][:rule]
#--> no-aggressive-verbs
? aFindings[1][:message]
#--> aggressive verb in method 'KillOrder' -- prefer Remove/Delete/Dispose/Clear/Close (Softanza
tone)
? StzCodeIsClean(cCode)
#--> 1
```

✓ exécuté : a affiché ce que ce bloc promet

Lisez deux fois la dernière ligne. `StzCodeIsClean` répond encore `1` : le code est propre ! La règle de ton avertit ; elle ne refuse pas. Une règle de mots conseille. La doctrine du domaine de projets dit quelque chose de proche de sa propre loi des mots : aucune session ne peut tester l'effet d'un mot sur son lecteur, si bien que cette loi est tranchée plutôt que vérifiée. Le message dit `aggressive verb in method 'KillOrder'` et nomme le remède, `prefer Remove/Delete/Dispose/Clear/Close`. Que dit la règle une fois que le verbe est `Remove` ? Exécutez-le :

HARO

```
cCode = "class Shop" + nl + "def RemoveOrder()" + nl + "  return" + nl
? len( StzCheckCode(cCode) )
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Aucun constat ne revient : `0`. Une fois que le verbe est `Remove`, la règle n'a rien à dire.

Et les règles forment un ensemble fermé. Combien y en a-t-il, et un agent peut-il les lister avant d'écrire ?

HARO

```
? len( StzCodeRuleNames() )
#--> 14
? StzCodeRuleNames()[3]
#--> no-aggressive-verbs
```

```
✓ exécuté : a affiché ce que ce bloc promet
```

Oui : quatorze noms, et le troisième est la règle qui vient de parler. Le dossier ne relie pas la règle de ton de la bibliothèque à la loi des mots du domaine de projets, tranchée le 21 août 2026, je ne le ferai donc pas non plus. Mais c'est le même instinct à deux altitudes : les mots que nous employons à propos du travail font partie du travail.

Voilà les lois. Reste ce que j'ai refusé, et la perte que j'ai acceptée.

LA DÉCISION, ET CE QUI A ÉTÉ REFUSÉ

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé un cadre logiciel. L'arrangement, ce sont des fichiers, un historique et quelques hooks. Un superviseur qui planifierait les agents d'IA serait un programme de plus à qui se fier, et une chose de plus qui décide sans trace. Ici, une session qui ne tourne pas ne laisse que ses fichiers, et les fichiers sont la vérité.

J'ai refusé qu'un coordinateur note son législateur. L'autorité descend de moi ; la correction remonte, et la session centrale est corrigée par l'Observateur, un bureau qui audite la façon dont le harnais est construit et ne répare rien lui-même, et liée par la doctrine, les lois écrites du domaine de projets. Un coordinateur qui jugerait ses propres règles serait la seule partie du système que rien ne pourrait corriger. Je tiens cela pour une vue, non pour un résultat.

J'ai refusé d'automatiser au maximum. À partir du milieu d'août 2026, des sessions ont été ouvertes par des horloges. Le 27 août, je les ai arrêtées. Le dossier garde ma raison telle qu'elle a été écrite alors : les routines consommaient du quota sans valeur ajoutée. Et il garde la partie encore ouverte : ce jugement était la seule mesure que quiconque avait, parce que la ligne de coût que chaque session écrit à sa clôture ne porte aucun compte de jetons. Le harnais ne dit pas à une session ce qu'elle a dépensé. Sur les trois cent quatre-vingts lignes de coût écrites jusqu'ici, une seule porte un chiffre ! Qu'ont coûté les horloges ? Ce n'est pas dans le dossier : il garde la raison et aucun chiffre, et je n'en donne aucun. Depuis, la décision est qu'une minuterie peut lancer un script, jamais un modèle d'IA, et toutes sont en pause pour l'instant. La direction est qu'une session démarre quand un fichier arrive dans un dossier surveillé, ou quand je la démarre ; le surveillant est conçu et reste à venir (section 7).

J'ai refusé de trancher en mon absence. Une décision n'est écrite dans le dossier que tant que je suis là. Le 28 septembre, j'ai demandé à la session centrale de régler ce qui était en attente à ma place. Les cinq décisions qu'elle a prises ce jour-là

sont marquées comme prises sur ma délégation. La marque est là pour que personne ne les prenne plus tard pour les miennes. Les décisions prises pour ce livre sur ma délégation portent la même marque.

J'ai refusé que le protocole devienne un produit. La doctrine du harnais est ouverte, et elle ne doit jamais être vendue : sa valeur est dans ce qu'elle refuse. Elle n'a été prouvée jusqu'ici qu'en un seul endroit : une personne, une machine, un système de fichiers, une quinzaine de sessions. Le premier adoptant hors de ce domaine de projets est l'étape suivante.

Et j'ai accepté une perte. Le code des deux dernières années a été écrit en grande partie par des sessions, et ma main, qui était le style, s'est amincie là où elles ont écrit. Mon étude du dossier l'a mesuré. La promesse sous chaque ligne (la ligne `#-->`) est la manière de la maison d'écrire un exemple. Elle se tient à près de cent pour mille lignes dans les recettes courtes (les quickers) et à soixante-quinze dans mes articles. Dans les gardes narrées que les sessions ont écrites, qui affirment sous une autre forme, elle se tient à moins de un. Les sessions ont même pris le nom de mes articles pour leurs fichiers de test, un nom qu'un modèle a proposé. J'accepte le code d'implémentation qu'elles ont écrit. Je n'accepte pas que le style qu'un programmeur ou un agent d'IA écrit avec Softanza soit le leur plutôt que celui de la maison. C'est pourquoi la loi de la partie II existe, et pourquoi ce livre est écrit de ma voix, avec les sessions comme chercheuses et comme tribunal. Et c'est pourquoi chaque chapitre attend ma lecture avant de perdre le mot brouillon.

Vient ensuite le tribunal, et ce qu'il ne peut pas prouver.

5. LE TRIBUNAL

Aucune garde de la bibliothèque ne prouve le protocole. Ses tribunaux sont hors de la bibliothèque, et je les nomme par ce qu'ils font. Un hook refuse l'indexation en bloc, et un autre refuse le travail quand la machine n'a plus de mémoire à revendre. Une vérification dit si les hooks installés sont les hooks écrits. Un instrument

compare chaque horodatage à son commit et publie les lignes. Ce sont des scripts, pas du Haro, et le lecteur de ce livre ne peut pas compter leurs assertions ; ils sont donc nommés ici et non dans la ligne du tribunal ci-dessous.

Les blocs de ce chapitre reposent sur les gardes de la bibliothèque pour ses règles de la maison. `coderule_narrated` prouve qu'une règle de code est une règle sur le graphe, et que les constats de la maison sortent sur une fixture plantée sous une forme figée. Elle prouve que les règles sont au nombre de quatorze, et qu'une définition ou un commentaire n'est pas pris pour un appel. `codegraph_narrated` prouve le graphe en dessous, et `coderule_project_narrated` lance les règles de projet sur un dossier de fixture.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`coderule_narrated` 77 assertions
base/test/meta/coderule_narrated.ring

`codegraph_narrated` 28 assertions
base/test/meta/codegraph_narrated.ring

`coderule_project_narrated` 13 assertions
base/test/meta/coderule_project_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en échelle et en indépendance. Ses cadres logiciels tournent sur de nombreuses machines pour de nombreuses équipes ; ce protocole est prouvé jusqu'ici chez une personne, sur une machine, un système de fichiers, et le premier essai hors du domaine est l'étape suivante. Sa doctrine le dit dans ses propres mots : ce sont des lois qui ont survécu à un seul domaine.

Le terrain mène en comptabilité. Une interface payante rapporte les jetons de chaque appel. Le dossier des coûts de ce domaine de projets ne porte pas encore de compte de jetons ; l'ajouter est l'étape suivante, et d'ici là la décision économique la plus importante qu'il ait prise l'a été au jugement.

Le terrain mène en outillage pour de nombreux auteurs. Les branches, les files de fusion et les propriétaires de dossiers traitent la concurrence à une échelle que le domaine de projets n'a pas rencontrée. Ici, la réponse est une discipline et un hook. Le hook lit le texte d'une commande, si bien qu'il refuse aussi un message de commit qui ne fait que citer la forme interdite. Ce coût est connu et accepté.

Là où je pense que le dossier se distingue, c'est dans une habitude : chaque loi porte l'échec qui l'a payée, et le dossier publie son propre taux d'infraction à sa propre loi. Le bureau central a enfreint la loi des horodatages à peu près aussi souvent que le domaine de projets qu'il mesurait, sur une surface différente : 5 sur 25 contre 34 sur 151. Quatre des cinq datent d'avant la loi. Les deux chiffres sont imprimés côte à côte.

Reste la limite, en mots d'étape.

7. LA LIMITE

CONSTRUIT — le protocole, ses mémos et son registre des décisions ; trois hooks : l'un refuse l'indexation en bloc, l'autre refuse une modification de registre faite sur une vue périmée de son système source, le troisième refuse le travail sur une machine inondée ; l'instrument qui mesure les horodatages composés ; les quatorze règles de la maison de la bibliothèque, dont cinq sur un graphe du code

CONÇU — un surveillant qui réveille une session quand un fichier arrive, à la place des horloges ; un tribunal de style pour le code qu'un programmeur ou un agent d'IA écrit

ABSENT — des chiffres de jetons dans le dossier des coûts ; une équipe hors de ce domaine de projets qui ait essayé le protocole ; une vérification d'un mot dans la langue du lecteur

Ce qui est construit : le protocole et ses fichiers, les mémos datés, le registre des décisions, trois hooks qui s'exécutent dans chaque session sur cette machine, et l'instrument qui mesure les horodatages. Les quatorze règles de la maison de la bibliothèque sont construites, dont cinq sur son graphe de code.

Ce qui est conçu : un surveillant qui réveille une session quand un fichier arrive, ce qui est, à mon avis, le bon remplacement des horloges et reste à venir. Et un tribunal de style pour le code qu'un programmeur ou un agent d'IA écrit avec la bibliothèque. Le style de la maison serait alors tenu par une machine autant que par ma lecture.

Ce qui est absent : un chiffre de jetons dans le dossier des coûts ; toute équipe hors de ce domaine de projets qui ait employé le protocole ; et une vérification qu'un mot se lit bien dans la langue du lecteur. La loi des mots demande cette vérification, et aucun outil ne l'exécute encore ; cette vérification est l'étape suivante. Mon étude du dossier a nommé une limite de plus : le protocole qui rend le dossier digne de confiance coûte aussi de l'énergie dont le produit a besoin, et je devrais surveiller le rapport. Le dossier ne contient aucune lecture de ce rapport.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Indexez par chemin explicite, relisez un fichier avant de l'écrire, lisez l'heure sur l'horloge. Chaque règle a coûté une modification à quelqu'un.
L'analyste	Le dossier est une donnée que vous pouvez interroger, et il publie ses propres taux d'erreur à côté de ses règles.
Le concepteur	Un mot nomme un état, jamais une personne. Un message est une structure fermée qu'un lecteur parcourt d'un coup d'œil, écrite pour rester lisible à un lecteur dont la première langue n'est pas l'anglais.
Le décideur	Une décision n'est écrite qu'en votre présence. Quand aucun chiffre ne peut répondre à une question, décidez par le jugement et écrivez que c'était un jugement.
Le formateur	Enseignez une règle avec l'échec qui l'a gagnée. Une règle dont vous ne pouvez pas nommer le coût est une règle dont vous vous dispenserez à deux heures du matin.
L'agent	Vous êtes un ouvrier avec un dossier. Vous écrivez un mémo, une ligne de coût et une conclusion ; les hooks refusent ce qui a fait du mal à d'autres ; vos horodatages sont comparés à vos commits. Rien de cela ne suppose que vous êtes négligent. Tout cela suppose que vous êtes nombreux.

9. EXERCICE

Prenez les dix derniers messages, notes ou lignes de journal où vous avez écrit une heure à la main. Pour chacun, trouvez le moment que le système a enregistré de lui-même : le commit, l'heure d'envoi du message, l'heure d'enregistrement du fichier. Avant de regarder, notez combien de vos heures vous vous attendez à voir en retard sur celles du système d'une minute ou plus. Puis comptez.

Votre compte est le constat. Je ne vous donnerai pas le mien.

Je vous laisse celui-ci. Prenez le premier bloc de la section 3, celui qui demande à la bibliothèque de vérifier `KillOrder`, et retirez ses promesses : le bloc ci-dessous est celui-là, sans aucune ligne `#-->`. Avant de le lancer, écrivez sous chaque question la ligne que vous attendez. Puis lancez-le et comparez. La section 3 dit pourquoi la dernière réponse est ce qu'elle est ; si c'est la réponse que devrait donner une fonction de ce nom, je vous le laisse.

HARO

```
cCode = "class Shop" + nl + "def KillOrder()" + nl + "  return" + nl
aFindings = StzCheckCode(cCode)
? aFindings[1][:rule]
? aFindings[1][:message]
? StzCodeIsClean(cCode)
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Quand vous aurez tranché, vous saurez pourquoi une règle de mots conseille et ne refuse pas.

Le dernier chapitre porte sur la ligne suivante. Son principe dit quand cette ligne pourra être écrite : le dossier ne porte une solution que lorsqu'elle a tourné sur son propre moteur et qu'une personne nommée a dit ce qu'elle a vu.

> **OÙ SONT LES PREUVES**
>
> | | |
> |---|---|
> | NARRATION | `stz-rules-as-graph-rules-narration.md` |
> | NARRATION | `stz-governance-lineage-narration.md` |
> | SUR LE SITE | estate |
> | SUR LE SITE | principles |
> | DÉCISIONS DE L'ÉTUDE | 12.16, 12.20, 12.29 |

PARTIE VII · CHAPITRE 33 · COLONNE

La ligne suivante : une solution sur son propre moteur

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:01 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 3 bloc(s) sur 3 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

Dans ce chapitre, je vous montre le plancher sur lequel la ligne suivante se tiendra. Une plateforme est déclarée, construite et déployée dans cet ordre ; un jumeau ne touche rien de réel ; une dernière porte refuse un faux service. Puis je dis ce que les trois projets établissent aujourd'hui. Je lis ensuite dans le dossier les cinq pas qui mènent de ce plancher à la ligne suivante : une solution sur le moteur propre de Softanza, et ce que dit une personne nommée qui l'a utilisée. Je dis cinq choses que j'ai refusées, dont une référence sans document, mon propre chiffre pour le runtime de pont, et un instrument tenant lieu de preuve d'une perception. Le dossier ne porte une solution que lorsqu'elle a tourné sur son propre moteur et qu'une personne nommée a dit ce qu'elle a vu.

1. LE CONCEPT, TEL QUE LE TERRAIN LE TIENT

Dans notre industrie, une plateforme est prouvée quand quelqu'un d'autre la fait tourner pour de bon. La page d'accueil montre les logos des entreprises qui l'utilisent ; une étude de cas en raconte une en deux pages. Une équipe qui croit à son propre produit l'utilise pour son propre travail, et un compilateur sérieux se compile lui-même. Les ingénieurs qui ont besoin d'une échelle en ont une : les niveaux de maturité technologique de la NASA, devenus plus tard une norme. Le niveau le plus haut y est un système éprouvé à l'endroit pour lequel il a été construit.

Ce sont des épreuves raisonnables, et elles pointent toutes vers la même chose : la production. Ce qu'elles ne demandent pas, c'est qui l'a vu fonctionner, et ce que cette personne a dit.

2. LE PREMIER PRINCIPE

Le dossier ne porte une solution que lorsqu'elle a tourné sur son propre moteur et qu'une personne nommée a dit ce qu'elle a vu.

3. LA FORME SOFTANZA

Ceci est le dernier chapitre, et il regarde devant. Laissez-moi d'abord vous montrer le plancher sur lequel la ligne suivante se tiendra, car le plancher existe et tourne.

Une solution est déclarée avant d'être construite, et la plateforme qui la porte refuse d'être déployée avant d'être construite. Que répond-elle quand vous lui demandez de déployer d'abord ? Lancez-le :

HARO

```
oSol = new stzPlatformProfile("bank")
oSol.DevelopedOn(:Windows)
oSol.WithServer(:backend, :LinuxServer)
oPlatform = StzPlatformQ("bank")
oPlatform.SetProfile(oSol)
try
        oPlatform.Deploy()
catch
        ? cCatchError
done
#--> stzPlatform.Deploy: the platform must be Build() before it can be deployed.
oPlatform {
        ? IsDeployed()
        #--> 0
        Build()
        Deploy()
        ? IsDeployed()
        #--> 1
}
```

✓ exécuté : a affiché ce que ce bloc promet

La plateforme répond : `stzPlatform.Deploy: the platform must be Build() before it can be deployed.` Construite, puis déployée, `IsDeployed()` répond `1`.

Avant que quoi que ce soit soit réel, la solution peut tourner sur un jumeau, une copie du système qui ne garde aucune référence à la réalité. Où vit un fichier écrit dans le jumeau ? Lancez-le :

HARO

```
oTwin = new stzVirtualFileSystem()
oTwin {
        CreateFile("orders/today.txt", "3 couscous")
        ? ContentOf("orders/today.txt")
        #--> 3 couscous
        ? OriginOf("orders/today.txt")
        #--> virtual
}
? StzEngineFileExists("orders/today.txt")
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Il répond `virtual`, et `StzEngineFileExists` répond `0` : le fichier a été écrit, relu, et n'est pas sur le disque à cet emplacement !

Et à la barrière de production, une solution ne peut pas atteindre la production avec un faux service lié en elle, quel que soit celui qui le demande. Au chapitre 1 vous avez vu le registre refuser ; ici c'est la livraison elle-même qui refuse, avec une personne habilitée pour acteur. Que répond la livraison quand cette personne déploie en production avec un bac à sable lié ? Lancez-le :

HARO

```
oReg = new stzServiceRegistry("shop")
oReg.BindSandbox(:payments, new stzPaymentsSandbox())
oDel = new stzDelivery("shop")
oDel {
        AddBackend("api", "linux")
        UseServices(oReg)
        DeployTo(StzDeploymentSiteQ("host1"), "api")
        SetActor(HumanActor("dana"))
        oDep = Deploy(:Production)
        ? oDep.WasCommitted()
        #--> 0
        ? Log().EntriesOfLevel(:error)[2][:message]
        #--> sandbox-in-production @ shop/payments
}
```

✓ exécuté : a affiché ce que ce bloc promet

La personne avait le droit de déployer. La livraison l'a refusé ! `WasCommitted()` répond `0`. Pourquoi a-t-elle refusé ? La première entrée d'erreur du journal dit seulement que la surface externe n'est pas prête pour la production ; la seconde nomme le service : `sandbox-in-production @ shop/payments`. Un service de paiement qui est un bac à sable ne peut pas être ce par quoi un vrai client paie. Ce n'est pas une question d'autorité. C'est une question de ce qui est vrai.

Le plancher est donc là : déclarer, construire, répéter, refuser le faux, valider par un acteur gouverné.

Maintenant ce que les projets montrent, et l'étape qui les suit.

Softanza a-t-elle été utilisée ? Oui. Je l'ai écrit au chapitre 1 et je m'y tiens. Des parties de Softanza sont en production, et d'autres sont en développement. Trois projets, et leur retour, ont façonné sa conception. Softibank, une banque de Niamey, a un produit d'évaluation d'organisation que j'ai construit à côté de la bibliothèque, installé sur son propre réseau sous licence. Le guide d'installation remis à la banque l'atteste. Un restaurant de Lyon a reçu la livraison d'une première version d'une plateforme de commerce de quartier. Un avenant a été clos le 4 août ; le carnet montre que le propriétaire a pris part à la première démonstration, et le 19 août il a fait une journée de mesures sur le terrain avec deux téléphones. Le dernier enregistrement daté de ce projet est du 25 août 2026, et le prochain sera celui que la phase suivante écrira. Une organisation du Niger a son hub conçu. Le site énumère, projet par projet, chaque leçon qu'il nomme et le fichier de la bibliothèque qui la porte. La partie VI de ce livre raconte leurs histoires.

Et le site dit, dans la même page, sur quoi ces plateformes tournent aujourd'hui : des piles ordinaires. Celle de la banque est une application Python. Celle du restaurant est un ensemble de mondes web, chacun un seul fichier. Le hub de l'organisation est conçu, et sa construction est à venir. Le site dit que leurs déclarations sont écrites comme celles de Softanza elle-même, pour que les faire passer sur le moteur Softanza soit un changement de moteur et non de plan. Les mots du site sont exacts : c'est un pas en avant, pas un pas accompli. Ce pas est le prochain.

La ligne suivante est donc celle-ci : *la solution a tourné sur le moteur propre de Softanza, et voici ce que l'utilisateur a dit.*

Qu'est-ce qui mène d'ici à cette ligne ? Cinq pas, lus dans le dossier et non composés.

1. **Les déclarations d'un client tournent sur le moteur au lieu de la pile ordinaire.** Le site dit que les déclarations sont en cours d'écriture, comme celles de Softanza elle-même ; je n'en ai trouvé aucun fichier pour l'instant, et le chapitre 31 dit où j'ai cherché. Le passage au moteur est l'étape suivante.
2. **Le runtime propre à la langue fait tourner la bibliothèque sur le système du client.** Aujourd'hui les blocs de ce livre sont exécutés par le runtime de l'ancien langage hôte, version 1.27, et chaque page le dit sous sa ligne de preuve. Le runtime de pont censé le remplacer a été signalé le 5 octobre 2026 comme passant toutes les assertions que l'ancien runtime passe sur la bibliothèque, sauf une requête HTTP/2. Cette requête a échoué une fois dans la passe et passe deux fois quand on la lance seule. Sur chaque ligne stable, 3 461 des 5 112 programmes concordent avec l'ancien runtime. Ce sont deux comptes et non un seul : le premier porte sur les assertions de la suite, le second sur des programmes entiers, un programme n'étant compté comme concordant que si chaque ligne stable qu'il imprime est la même sur les deux runtimes. La passe du matin en donnait 3 425, le chiffre que le chapitre 27 imprime ; les 3 461 sont ceux du soir, après une journée de corrections. Son passage sous Linux est la prochaine étape. La charte de Haro a été ratifiée le 1er octobre 2026, et les étapes propres de Haro passent sur ses propres fixtures. Que le runtime propre de Haro fasse tourner la bibliothèque n'est pas encore mesuré.
3. **La solution écrit sa fiche de garanties.** La vision demande à chaque solution cinq lignes, une pour chaque système auquel elle répond, technique, social, écologique, économique et culturel, jugées comme des fixtures, la machine en étant dérivée. La fiche est conçue ; la première reste à écrire.
4. **Des hôtes réels et une vraie carte.** La vérification du déploiement contre un hôte réel ou un compte réel dépend de l'infrastructure. La machine déclarée, un appareil écrit comme une déclaration, démarre et se juge elle-même dans un émulateur. Son profil hébergé fait quatre promesses : être toujours joignable, avoir un nom stable, tenir un journal durable, et survivre à une coupure.

L'émulateur peut en vérifier trois. La quatrième attend un chien de garde sur une vraie carte, un chien de garde étant, au sens courant du terrain, une minuterie matérielle qui redémarre une machine qui ne répond plus. Le site dit qu'aucune charge de production n'y est encore.

5. **Une personne le perçoit, et le dossier dit qui.** Pas un compteur. Une personne, nommée, avec une date et ce qu'elle a dit.

Le dernier pas est une loi du domaine de projets, et elle a été gagnée. Le volet sonore de la bibliothèque a un jour tenu des centaines d'assertions réussies alors qu'à mon oreille il ne sonnait pas encore juste. Depuis, un travail qui atteint une personne par un sens doit être livré sous une forme qu'une personne peut percevoir, et le dossier doit dire qui l'a perçu et ce qu'elle a dit. Et quand personne ne l'a fait, la loi le dit clairement : non perçu est un état, pas un échec. Un rapport peut être honnête et rester non perçu.

Le dossier tient déjà la forme de la ligne. Dans un autre volet, le 2 octobre 2026, le dossier dit que le Principal, son nom pour moi (le chapitre 32 le dit), a écouté un son que la bibliothèque avait fait et a dit, dans les mots écrits : « yes it's real drum!! » (les mots sont écrits en anglais dans le dossier) Ce n'est pas une solution, et ce n'est pas un client. Mais c'est une personne, une date et une phrase, et c'est à cela que ressemblera la ligne suivante.

Le dossier tient aussi deux verdicts à venir. Les images que la bibliothèque dessine ont été jugées par un modèle d'IA, quarante et une images ou sections de texte, vingt justes, neuf fausses, douze incertaines. Le dossier dit qu'aucun humain ne les a encore regardées ; ce regard est le prochain verdict. L'exécution en réel du volet des paiements avec le compte de test de l'organisation a été reportée, par moi, et rapportée comme non perçue. Ni l'un ni l'autre n'est un échec. Les deux sont la suite.

4. LA DÉCISION, ET CE QUE J'AI REFUSÉ

J'ai refusé qu'on me dise que les projets ne comptaient pas. La première lecture du dossier a écrit qu'aucun utilisateur hors de mes propres projets n'avait été mesuré. C'était faux, et je l'ai dit : Softanza a été utilisée dans de vrais projets qui, sans être achevés, ont donné le retour qui a façonné sa conception. La lecture a été corrigée dans le dossier, sur place, avec la date. Ce qui l'a remplacée est exact : les utilisateurs existent, et leur retour a façonné la conception, et l'étape suivante est d'en passer un sur le moteur.

J'ai refusé une référence sans document. Le site ne compte un projet comme référence que lorsqu'un document le prouve, et il ne cite aucun nombre de pays ni aucun nombre de programmeurs. Une leçon n'est listée que lorsqu'un fichier de la bibliothèque la porte.

J'ai refusé de déplacer une plateforme en la réécrivant. La plateforme d'un client passe sur le moteur comme un changement de moteur, non de plan. Les déclarations sont à écrire une fois, comme celles de Softanza elle-même, et le moteur en dessous change. Je tiens cela pour une vue, non pour un résultat.

J'ai refusé mon propre chiffre. J'ai dit à voix haute, en pourcentage, jusqu'où le runtime de pont était arrivé. Je ne l'imprime pas sans la garde qui le produit. Le mot d'étape est « en construction », et les lignes mesurées sont la preuve.

J'ai refusé qu'un instrument tienne lieu de preuve d'une perception. Une phase dont la preuve est entièrement un compteur n'est pas achevée quand la chose elle-même est quelque chose qu'une personne voit ou entend. Le compteur est un résultat. La personne est le verdict. Le volet sonore plus haut, avec ses centaines d'assertions réussies et le son qui n'était pas encore juste, en est le fondement.

5. LE TRIBUNAL

Le plancher de ce chapitre repose sur trois gardes. `full_scope_model_narrated`, le socle commun de l'architecte, prouve qu'une solution est d'abord modélisée, qu'un déploiement avant une construction est refusé et consigné comme une erreur, et qu'un modèle d'IA ne peut pas effectuer le déploiement. `virtual_system_twin_narrated` prouve que la répétition ne touche rien de réel, que le plan se narre lui-même, et que valider est l'unique porte. `service_delivery_narrated` prouve qu'un déploiement en production avec un faux lié ne s'exécute pas du tout, avec chaque raison nommée, et que la barrière s'ouvre une fois le service réel.

Le tribunal du sujet de ce chapitre n'est pas une garde. C'est une ligne du dossier : qui, quand, et ce qu'il a dit. Cette ligne est la prochaine à écrire.

Lu dans l'arbre de tests de la bibliothèque au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Le chiffre est le nombre d'assertions que la garde contient, compté dans son fichier ; le site montre chacune exécutée.

`full_scope_model_narrated` 62 assertions
base/test/system/full_scope_model_narrated.ring

`virtual_system_twin_narrated` 48 assertions
base/test/system/virtual_system_twin_narrated.ring

`service_delivery_narrated` 60 assertions
base/test/system/service_delivery_narrated.ring

6. LE TERRAIN, ET OÙ IL MÈNE

Le terrain mène en production. Les piles ordinaires font tourner de vraies charges pour de vrais utilisateurs aujourd'hui, un fait du terrain et non une lecture du dossier ; la page même du dossier nomme parmi elles l'application Python de la

banque, installée sous licence. La première version du restaurant est faite de mondes web d'un seul fichier, livrée, avec une journée de mesures sur le terrain le 19 août, et le hub est conçu.

Le terrain mène en infrastructure réelle. Ses outils pilotent de vrais fournisseurs et de vrais hôtes chaque jour ; le déploiement de Softanza est éprouvé de bout en bout sur des sites locaux et répète le reste ; la vérification en réel contre un hôte réel est l'étape suivante.

Le terrain mène sur le matériel en service. Ses appareils tournent dans des boutiques et des cliniques ; la machine déclarée démarre dans un émulateur, et sa carte est l'étape suivante.

Le terrain mène en vitesse. Le runtime auquel Softanza s'est mesurée était plus rapide sur chaque ligne, d'un facteur compris entre un et deux, et la note du domaine de projets a consigné la mesure.

Là où je pense que le dossier se distingue, c'est dans ce à quoi il se tient. Chaque référence a un document. Chaque étape a un mot. Selon le document de conception, le déploiement émulé fait tourner le même artefact que celui qui est livré. Un faux ne peut pas être livré et un modèle d'IA ne peut pas valider. Et une perception est une ligne nommée, datée, citée, ce qu'un logo sur une page d'accueil ne porte pas.

7. LA LIMITE

CONSTRUIT	une plateforme déclarée construite, puis déployée, et refusée dans l'autre ordre ; le jumeau qui ne touche rien de réel ; la barrière de production qui refuse un faux service
CONÇU	le runtime propre à la langue, qui fait tourner la bibliothèque ; la fiche de garanties d'une solution en cinq lignes ; des hôtes réels et une vraie carte
ABSENT	une solution de client qui tourne sur le moteur Softanza, et le verdict d'une personne nommée à son sujet ; les déclarations des clients en fichiers

Ce chapitre regarde devant, la limite est donc courte.

Ce qui est construit : le plancher. Déclarer, construire, déployer dans l'ordre ; répéter sur un jumeau ; refuser un faux à la barrière de production ; valider seulement par un acteur gouverné.

Ce qui est conçu : le runtime propre à la langue, qui fait tourner la bibliothèque ; la fiche de garanties d'une solution en cinq lignes ; la vérification sur des hôtes réels et sur une vraie carte.

Ce qui est absent : une solution de client qui tourne sur le moteur Softanza, le verdict d'une personne nommée à son sujet, et les déclarations des clients en fichiers (le chapitre 31 dit où j'ai cherché). À côté se tiennent le plancher ci-dessus et le retour de trois projets. Le dossier ne nomme pas encore le projet qui passe en premier sur le moteur, ni la personne qui le percevra ; je laisse les deux non écrits jusqu'à ce que le dossier puisse les porter.

8. SIX PORTES

PORTE	CE QUE VOUS RETIREZ DE CE CHAPITRE
Le programmeur	Votre code tourne aujourd'hui par la construction, la répétition et un déploiement gouverné. Ce qui vient ensuite, c'est qu'un client le fasse tourner sur le moteur, et le dossier portera cette affirmation le jour où elle sera vraie.
L'analyste	Demandez deux choses avant de compter un projet comme preuve : un document, et un verdict nommé.
Le concepteur	Ce que vous concevez est jugé en dernier par le sens d'une personne. L'opinion d'un modèle d'IA sur une image est un résultat ; celle d'une personne est le verdict.
Le décideur	Lisez le mot d'étape avant la promesse. Aujourd'hui le plancher est construit ; l'exécution chez un client sur le moteur est le prochain jalon, et son mot d'étape est absent.
Le formateur	Enseignez que non perçu est un état et non un échec. Un rapport honnête peut être livré dessus, s'il le dit.
L'agent	Vous pouvez répéter un déploiement et l'expliquer. Vous serez refusé quand vous essaierez de l'effectuer (le chapitre 24 montre le refus).

9. EXERCICE

Déclarez une plateforme pour un projet qui est le vôtre, avec un serveur. Avant de lancer quoi que ce soit, écrivez la ligne que vous attendez sous chacune de trois questions : ce qui se passe quand vous lui demandez de déployer avant d'avoir construit ; ce qui se passe quand un assistant demande de déployer après la construction ; ce qui se passe quand une personne le demande.

HARO

```
oSol = new stzPlatformProfile("my-project")
oSol.DevelopedOn(:Windows)
oSol.WithServer(:backend, :LinuxServer)
oP = StzPlatformQ("my-project")
oP.SetProfile(oSol)
# your three questions here, each with the line you expect under it
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez-le. Puis celui-ci, que je vous laisse aussi. Voici le bloc de la barrière de production de la section 3, avec une seule ligne changée : l'acteur n'est plus la personne mais l'assistant, `LLMActor("assistant")`, comme au chapitre 24. Je n'ai pas lancé cette version, et je n'imprime aucune réponse pour elle.

HARO

```
oReg = new stzServiceRegistry("shop")
oReg.BindSandbox(:payments, new stzPaymentsSandbox())
oDel = new stzDelivery("shop")
oDel.AddBackend("api", "linux")
oDel.UseServicesQ(oReg)
oDel.DeployTo(StzDeploymentSiteQ("host1"), "api")
oDel.SetActor(LLMActor("assistant"))
oDep = oDel.Deploy(:Production)
? oDep.WasCommitted()
? oDel.Log().EntriesOfLevel(:error)[2][:message]
```

✎ non exécuté : un exercice, écrit pour que vous l'exécutiez

Lancez celui-là aussi. Lisez ce que répond `WasCommitted()`, et lisez le journal : le bloc de la section 3 imprimait sa deuxième entrée d'erreur pour la personne, et je ne dis rien de celle de l'assistant. La section 5 dit ce que prouvent les trois gardes, et la

section 8 ce qu'on refuse à un agent d'IA. Quand la barrière répondra, vous saurez ce qu'elle refuse pour ce qui est vrai et ce qu'elle refuse pour qui demande.

Chaque bloc de ce livre est à vous pour le relancer. Écrivez maintenant une ligne de plus, la suivante de votre propre projet, dans la forme du dossier : qui l'a perçu, à quelle date, et ce qu'il a dit. Si personne ne l'a fait, écrivez « non perçu », et dites pourquoi. Cette ligne est la fin de ce livre, et le début du suivant.

> OÙ SONT LES PREUVES
>
NARRATION	`stz-built-on-real-world-need.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
NARRATION	`stz-emulating-the-whole-solution-narration.md`
NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
SUR LE SITE	customers
SUR LE SITE	forged
SUR LE SITE	platforms
SUR LE SITE	estate
DÉCISIONS DE L'ÉTUDE	12.5, 12.10, 12.13, 12.21, 12.29, 12.30

Annexes

ANNEXE A

Le registre des innovations

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

Mon étude du dossier a lu quatre inventaires que la bibliothèque tient d'elle-même : les narrations avec leurs titres, les documents de conception, la phase des moteurs de paradigme du moteur, et la liste de concepts repensés d'un lecteur extérieur. Elle les a croisés, et a trouvé quarante-six innovations en sept familles. Voici ce registre, tel que l'étude l'a tranché le 7 octobre 2026, avec une colonne propre à ce livre : le chapitre qui raconte chaque ligne. Une décision ultérieure du même jour ajoute une quarante-septième ligne, la réponse par l'exemple, avec les mots d'étape d'un plan : rien n'induit encore, comme le dit le chapitre 18. Elle n'est pas dans ce tableau, qui garde les quarante-six que l'étude a comptées.

Lisez le registre comme l'étude le lit, et pas mieux. C'est une lecture et non un recensement. Un concept sans narration ni document de conception qui lui soit propre n'a pas été compté. Les preuves de l'étude comptent les fichiers qui nomment la chose dans du code hors des tests et des archives, les fichiers de test, et le module du moteur quand il en existe un ; elles comptent les fichiers qui nomment une chose, et non des tests qui passent. Je n'en ai recompté aucun. L'étude a aussi compté combien de pages du site nommaient chaque innovation ; le site a changé depuis, et je laisse cette colonne de côté.

Sur les quarante-six, huit ne sont pas racontées dans ce livre : le moteur des adverbes, les variables nommées, le find unifié, l'historique des objets, l'API des fichiers par intention, la navigation naturelle dans les dossiers, l'invisible et le sosie, et les statistiques avec le solveur linéaire et les matrices. Elles sont dans le registre parce que le dossier les tient, et non racontées ici parce que ce livre a dû choisir.

A · PARADIGMES

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE (7 OCTOBRE 2026)
1	La programmation orientée portée	un comportement avec un cadre invisible qui le gouverne, soigné en quatre gestes ; quatre instances	8, 23	une conception de 450 lignes ; des instances dans regex, système, livraison
2	L'échelle du Softanzuter	motif, Xuter, Softanzuter, agent d'IA	9, 17	deux moteurs, quinze blocs, un esprit d'agent
3	Reaxis	la programmation réactive repensée en Container, Stream, Rfunction	9	dix fichiers de code, cinquante-neuf fichiers de test, deux modules du moteur
4	La programmation presque naturelle, NNL et `stzNatural`	assez proche du langage naturel pour être lisible, assez structurée pour s'exécuter	13	trois narrations, deux documents de conception, un dossier de treize fichiers
5	La programmation par la connaissance, `.zknw`	la sémantique de la pensée : un fichier de connaissance, un système opérationnel	14	une narration qui s'exécute ; le graphe de connaissance en soixante et onze fichiers
6	Les fonctions comme expressions de la langue	vingt-deux formes qui encodent la sémantique du calcul	8	environ six mille étiquettes
7	Les quatre métaphores	le marcheur, le vérificateur, le producteur, l'exécutant	6	trois narrations
8	Le code conditionnel, W et WF	une condition voyage dans un argument	6, 12	une narration
9	Les actions futures	planifier d'abord une suite, puis l'exécuter d'un seul coup avec un rythme naturel	8 (retirées exprès, le 10 juillet 2026)	une narration de soixante-quinze lignes ; deux fichiers de test
10	La programmation centrée sur le raffinement	des points de raffinement, une barrière en quatre étapes, un fichier de raffinements	10 (quatre étapes dans la conception, cinq contrôles dans le code)	deux livraisons de la barrière de raffinement, avec leurs gardes

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE (7 OCTOBRE 2026)
11	Le wise coding	Softanza demande ; l'expression est libre, l'admission est gouvernée	18	deux gardes
12	L'intelligence programmatique, des agents qui ne peuvent pas vous nuire	l'esprit programmatique par défaut	17, 20	huit suites
13	Le modèle de langage du domaine, la prédiction constructive	la prochaine chose que dit un modèle est une unité vérifiable du monde	19	une garde ; un contrat

B · LES DOUZE MOTEURS DE PARADIGME

L'étude a compté douze moteurs ; le registre en donne neuf une ligne à eux, et la fin de cette page revient aux douze.

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
14	Le moteur de vérité	la vérité n'est pas binaire : une vérité configurable par domaine, la falsification sémantique	nommé au 3 ; son voisin, la chaîne de vérité, est au 13	trois fichiers de code, trois tests ; le moteur marqué terminé
15	Les quantificateurs probabilistes	peu, certains, la plupart, tous, avec des seuils configurables	3 (peu, certains et la plupart tirent au hasard, et je le dis)	cinq fichiers de code, treize tests ; le moteur marqué terminé
16	Le moteur des adverbes	de rapide à rapidement : la morphologie pour `Explain()`	non raconté	un fichier de code, aucun test
17	La ligne du temps	un espace de travail temporel : points, intervalles, zones bloquées, raisonnement sous contraintes	9, 15	sept fichiers de code, soixante-neuf tests
18	Le navigateur de grille	un curseur à deux dimensions qui connaît ses directions et ses bords	9	dix fichiers de code, quarante tests

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
19	La fusion de sections	l'arithmétique des intervalles, trois stratégies de fusion	9 (le verbe que l'article présente est absent)	cinq fichiers de code, aucun test
20	Les opérations profondes	le parcours récursif par chemins des listes imbriquées	9	quatorze fichiers de code, trente-sept tests
21	Les variables nommées	des noms dynamiques ; des objets nommés	non raconté	un fichier de code, aucun test
22	Le pont polyglotte, EXCIS et ExterLib	orchestrer une symphonie polyglotte	16	quatre narrations, une conception de 1 986 lignes

C · MOTIFS ET RÉACTION

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
23	Les six langages de motifs	l'expression régulière élevée aux listes, aux nombres, aux tables, aux matrices, au temps et aux graphes	15	13 385 lignes ; huit narrations
24	RegexMaker et RegexData	une expression régulière déclarée, non écrite	15	deux narrations
25	Le find unifié	une seule recherche sur tous les supports	non raconté	une narration

D · EXPRESSION ET PERCEPTION

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
26	L'orientation visuelle : `viz`, les boîtes, l'art des chaînes, le système de tracés	là où l'art ASCII rencontre la visualisation de données	7 (le préfixe viz seulement)	soixante-dix-sept fichiers pour viz
27	L'invisible et le sosie	démasquer les caractères Unicode cachés et les sosies	non raconté	deux narrations
28	L'historique des objets	la trace de la chaîne	non raconté	une narration qui s'exécute

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
29	L'idiome des chaînes et des listes	des listes dans des chaînes, les petits opérateurs, les doublons, les constructeurs intelligents	7	dix narrations
30	L'API des fichiers par intention	au-delà des modes de fichier	non raconté	une narration de 552 lignes
31	La navigation naturelle dans les dossiers	l'état suit l'action	non raconté	trois narrations

E · DONNÉES, TEMPS ET RAISONNEMENT

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
32	La programmation du temps	l'heure, la date, la durée, la planification de capacité au calendrier	9, 15 (la ligne du temps et Timex seulement)	six narrations
33	`stzGraph` et ses organes	vingt-cinq notations, un planificateur, une requête, un organigramme	14, 29, 30	huit narrations
34	Les règles comme règles de graphe	des règles qui voient le tableau entier	29, 30	un plan de conception ; une narration
35	Les nombres repensés	une refonte globale des nombres, exacts par défaut	22	une conception de 3 167 lignes
36	Les statistiques comme langage de pensée, le solveur linéaire, les matrices	l'étude ne donne aucune ligne pour cette entrée	non raconté	quatre narrations

F · SYSTÈME, LIVRAISON ET EXPLOITATION

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
37	La fondation système et le jumeau virtuel	trois portées sur une même solution ; répéter, puis valider	23, 24	des gardes ; le moteur
38	Émuler, déployer, approvisionner ; du	toute la solution qui tourne avant d'être livrée	24	quatre narrations qui s'exécutent

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
	développement au déploiement			
39	La virtualisation des services	coder d'abord, s'abonner ensuite ; la production refuse un faux	24, 28	une narration ; le registre
40	Le plan de la performance	le chronomètre honnête, le profileur d'images, la boucle gouvernée, le jugement, la queue	21 (le profileur d'images n'est pas montré)	dix narrations
41	Le plan de la sécurité	l'événement, le registre, la sentinelle, l'exercice, la réponse, l'attestation	21	huit narrations ; une analyse d'incident de 815 lignes
42	La filiation de la gouvernance	responsable pour toujours	30	une narration
43	La porte d'entrée et les secrets	l'authentification et les identifiants, qui se masquent eux-mêmes	21 (les secrets ; la porte d'entrée comme simple événement)	deux narrations
44	La journalisation structurée, les sens du moteur, l'analyseur XML qui dit non, l'accès sûr aux listes de hachage	l'étude ne donne aucune ligne pour cette entrée	27 (les sens du moteur)	quatre narrations qui s'exécutent

G · LA BIBLIOTHÈQUE QUI SE CONNAÎT

#	INNOVATION	LA LIGNE DU DOSSIER	DANS CE LIVRE	LES PREUVES DE L'ÉTUDE
45	La stratégie de marquage de l'information et l'auto-documentation	chaque unité retrouvable, explicable, conversationnelle	7 (les étiquettes, et le fait qu'aucun tribunal ne les lit)	une conception de 238 lignes
46	La gestion de la locale et la culture	un logiciel de portée mondiale, à la manière de Softanza	22 (la perte : pas de couche de locale)	une narration qui s'exécute

CE QUE DIT LE REGISTRE

La lecture de l'étude, que je garde : ce que le site nomme est ce qui a une page de produit, un principe ou un domaine de l'Atlas. Ce qu'il laisse dans l'ombre est ce qui traverse les domaines. Chaque paradigme de la première famille sauf les lignes 4 à 8 et 10 à 12, que le site nommait déjà ; les douze moteurs sauf le pont polyglotte ; les langages de motifs en tant que famille ; l'idiome. Ce livre est en partie la réponse à cela : ses parties sont coupées à travers les domaines, et non le long d'eux.

L'étude a aussi réconcilié les douze moteurs. Le macroplan marque la phase comme terminée. Huit des douze ont un fichier source sous leur nom dans le moteur ; quatre n'en ont aucun, qui sont la vérité, les quantificateurs, le polyglotte et le polycode, les deux premiers annoncés terminés comme bibliothèques. Du côté de la bibliothèque, trois moteurs n'ont aucun fichier de test : les adverbes, la fusion de sections et les variables nommées. Deux des trois sont les lignes 16 et 21 ci-dessus, que ce livre ne raconte pas, et la troisième, la ligne 19, il la raconte comme une absence.

CE QUI N'EST PAS MESURÉ

Aucun code n'a été exécuté pour faire le registre. La colonne des preuves compte les fichiers qui nomment une chose, et les marques « terminé » des moteurs sont celles du macroplan. Savoir si la narration d'une innovation s'exécute une fois ses noms périmés réparés n'a pas été testé. Le chapitre d'une ligne, dans la quatrième colonne, est le chapitre qui la raconte, et une ligne qui dit « nommé au » est une ligne que le chapitre mentionne et ne raconte pas.

> OÙ SONT LES PREUVES
>
> DÉCISIONS DE L'ÉTUDE 12.19

ANNEXE B

Les quatorze règles de la maison

BROUILLON · TRADUIT DE L'ANGLAIS

Exécuté le 2026-10-15 13:01 UTC+01:00 dans la bibliothèque au commit 93d7a39bd : 2 bloc(s) sur 2 ont tenu leur promesse.

Les blocs ont été exécutés par le runtime de l'ancien langage hôte, version 1.27 ; Haro n'exécute pas encore la bibliothèque. Les binaires du moteur ont été construits dans une copie de ce commit, le plus récent le 2026-10-05 20:27 UTC+01:00 ; 1 changement(s) du moteur commité(s) après cette construction n'y sont pas.

La bibliothèque a un tribunal sur son propre code. Il lit une classe comme un graphe (cinq règles) ou comme du texte (neuf) et la juge selon des règles, dont certaines ont été écrites après qu'un défaut de cette forme a été trouvé dans le code même de la bibliothèque. Cette annexe énumère les quatorze, et en montre trois qui se déclenchent.

Il y a deux quatorze dans ce livre, et ce sont deux ensembles différents. Les quatorze **règles de la maison** de cette annexe sont des règles du code de la bibliothèque : elles forment un tribunal, et elles s'exécutent. Les quatorze **règles d'écriture**, au chapitre 7, sont une lecture de la manière dont le style s'écrit. Ce sont des lectures du code, énoncées comme des règles, et aucune machine ne les juge. La coïncidence du nombre est une coïncidence.

La liste des noms compte quatorze et la bibliothèque peut l'imprimer. Une passe de texte de plus, `for_in_loop`, rapporte aussi, de même que deux règles sur un projet entier, `no-dead-code` et, dans l'audit approfondi, `no-cyclic-calls` :

HARO

```
? len( StzCodeRuleNames() )
#--> 14
? StzCodeRuleNames()[2]
#--> q-returns-object
? StzCodeRuleNames()[14]
#--> value-called-as-function
```

✓ exécuté : a affiché ce que ce bloc promet

Voici trois d'entre elles, déclenchées sur du code que j'ai planté. Une méthode qui écrit sur la console est un avertissement, le code reste donc propre. Une méthode qui ne diffère d'une autre que par la casse est une erreur, puisque le langage hôte ne distingue pas les deux et que le fichier ne se chargerait pas. Et une classe qui définit `Len()` masque la fonction propre de l'hôte, et elle est refusée :

HARO

```
cSay = "class Door" + nl + "def Say()" + nl + " ? 'hello'" + nl + "     return 1" + nl
? StzCheckCode(cSay)[1][:rule]
#--> library_prints
? StzCheckCode(cSay)[1][:severity]
#--> warning
? StzCodeIsClean(cSay)
#--> 1
cBox = "class Box" + nl + "def Size()" + nl + " return 1" + nl + "def size()" + nl + "  return 2"
+ nl
? StzCheckCode(cBox)[1][:rule]
#--> no-case-collision
? StzCodeIsClean(cBox)
#--> 0
cLen = "class Box" + nl + "def Len()" + nl + "  return 1" + nl
? StzCheckCode(cLen)[1][:rule]
#--> no-len-method
? StzCodeIsClean(cLen)
#--> 0
```

✓ exécuté : a affiché ce que ce bloc promet

Une note sur l'orthographe, qu'un programme qui lit les constats doit connaître. La liste emploie des traits d'union : `library-prints`. Un constat de huit des quatorze revient avec des tirets bas : `library_prints`. Les constats des cinq règles sur le graphe du code (lignes 1, 3, 4, 5 et 6) gardent leurs traits d'union, et celui de la ligne 2 aussi. Je donne ce que l'exécution a imprimé.

LES QUATORZE

#	RÈGLE	CE QU'ELLE REFUSE	GRAVITÉ
1	`no-len-method`	définir `Len()` sur une classe : elle masque la fonction native de l'hôte et casse tous les appelants	erreur
2	`q-returns-object`	une méthode nommée avec un `Q` final qui ne renvoie pas un objet à chaîner	erreur
3	`no-aggressive-verbs`	les noms de méthode qui commencent par Kill ou Destroy : le ton Softanza	avertissement
4	`engine-first`	du code nouveau qui prend les fonctions de chaîne orientées octets de l'hôte là où les formes du moteur existent	avertissement
5	`q-has-plain-twin`	un mutateur chaînant sans jumeau simple : `Q` chaîne, et le simple agit	avertissement
6	`no-case-collision`	deux méthodes d'une même classe qui ne diffèrent que par la casse : l'hôte ne distingue pas la casse, le fichier ne se charge donc pas	erreur
7	`dead-knob`	un attribut qu'un setter écrit et que rien ne lit : un réglage qui ne peut rien changer	erreur ; avertissement quand seul son propre accesseur le lit
8	`setter-resets-on-reject`	un setter qui répond à une valeur qu'il n'aime pas en écrasant ce qui était déjà là	avertissement
9	`setter-only-moves-one-way`	`max()` ou `min()` contre l'attribut même qu'on règle : un cliquet qui avale l'autre sens	erreur
10	`misspelled-name`	une méthode portant une faute d'orthographe connue alors que le nom correct n'existe pas	avertissement
11	`library-prints`	une méthode de classe qui écrit sur la console pendant qu'elle travaille ; les méthodes dont le but est d'afficher sont exemptées	avertissement
12	`empty-method-body`	une méthode déclarée qui ne fait rien ; plus sévère pour un prédicat, qui alors ne répond rien, lu comme faux	avertissement ; erreur pour un prédicat
13	`writes-a-mutable-constant`	affecter à `true`, `false`, `null`, `nl`, `tab` ou `cr` : dans l'hôte ce sont des globales ordinaires	erreur

#	RÈGLE	CE QU'ELLE REFUSE	GRAVITÉ
14	`value-called-as-function`	appeler le résultat d'un appel, la forme que laisse un remplacement global de jetons	erreur

J'ai déclenché les lignes 11, 6 et 1 ci-dessus, dans cet ordre. Le chapitre 32 déclenche la ligne 3, le chapitre 7 les lignes 10 et 13, et le chapitre 5 la ligne 2. Pour les lignes 7, 9 et 12, j'ai lu les gravités dans la source, comme pour les lignes 4, 5, 8 et 14 ; je n'ai pas déclenché ces sept-là.

CE QUE LE TRIBUNAL NE JUGE PAS

Aucune des quatorze n'est une règle d'écriture. Aucune machine ne juge, aujourd'hui, le style d'un programme écrit avec la bibliothèque. C'est la limite du chapitre 7, et c'est la raison pour laquelle les quatorze autres sont dans ce livre sous la forme d'une table avec leur état, et non d'un tribunal.

LES QUATORZE AUTRES : LES RÈGLES D'ÉCRITURE, ET OÙ EN EST CHACUNE

Lu d'après la décision de l'étude sur le style, avec l'état de chacune au commit sur lequel ce livre tourne.

#	RÈGLE D'ÉCRITURE	ÉTAT
1	Une valeur est élevée à son objet par `Q()` ou par un élévateur typé	tourne
2	Une scène est un bloc entre accolades sur un seul objet	tourne
3	Chaque ligne imprimée porte sa réponse sous `#-->`	la promesse est jugée quand elle est écrite ; une ligne sans promesse n'est pas refusée
4	Un paramètre est nommé, et le nom se lit comme de l'anglais	tourne, et dans n'importe quel ordre
5	Une chaîne reflète l'ordre de la pensée, un pas par ligne	tourne
6	La forme du verbe dit ce qui arrive	tourne
7	Les suffixes et les préfixes sont petits et veulent dire chacun une seule chose	en partie

#	RÈGLE D'ÉCRITURE	ÉTAT
8	Une fonction a des variantes pour la manière dont une personne la dit	en partie
9	La précision plutôt que la généralité, quand cela ne coûte rien	en partie
10	Le code voyage dans un argument comme une petite chaîne	le siège tourne ; la marque sur le nom d'un paramètre est retirée
11	Les petits langages littéraux vivent dans des chaînes	tourne sous une forme
12	Une phrase peut être presque naturelle	pas comme le deck l'a écrit
13	Une fonction peut être nommée dans n'importe quelle langue humaine	les noms chinois et français lèvent une erreur ; l'arabe n'a pas été exécuté
14	Un commentaire parle avant le code, de la voix du maître	tourne

OÙ SONT LES PREUVES

DÉCISIONS DE L'ÉTUDE 12.16

ANNEXE C

Les vingt-deux verbes et la constitution

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

La constitution de l'interface vit dans son propre dépôt, et le site dit que ce dépôt n'est pas encore public. Ce livre l'appelle la constitution de l'interface et ne nomme pas le produit auquel elle appartient. La bibliothèque ne tient que la place de ses verbes et le chercheur des éléments qui ne servent aucun verbe ; la liste elle-même est celle de la constitution. Cette annexe imprime donc ce que j'ai lu dans le fichier même de la constitution le 9 octobre 2026 : sa version est 3.11, mise à jour pour la dernière fois le 16 août 2026, avec sept articles, trente et un ensembles et cent vingt-deux règles. Les nombres de cette phrase sont ceux du fichier, et je n'en ai compté aucun moi-même.

Le chapitre 26 montre comment la bibliothèque trouve un élément qui ne sert aucun verbe. Le chapitre 25 montre ce qu'un nouveau médium ajoute. Le chapitre 20 montre le droit à un humain comme la fin de la boucle.

LES QUATRE AXIOMES

Ses quatre axiomes sont le calme plutôt que la puissance, l'intention plutôt que la mécanique, le visible plutôt que le caché, le métier plutôt que la technologie. Son préambule dit que le logiciel de gestion n'est pas un divertissement, que son interface ne doit pas impressionner, et qu'elle doit servir.

LES SEPT ARTICLES

ARTICLE	TITRE	CE QU'IL CONTIENT
I	Calme	l'interface ne doit jamais concurrencer la pensée de l'opérateur ; le calme est une exigence fonctionnelle et non un choix esthétique
II	L'intention avant la mécanique	l'interaction part de ce que l'humain essaie de faire, et les widgets sont des conséquences de l'intention, jamais des primitives ; sélectionner n'implique jamais exécuter
III	État visible	si le système sait quelque chose d'important, l'utilisateur doit le voir : filtres actifs, brouillons en attente, normes, travail en arrière-plan, provenance machine
IV	Réversibilité	tout acte est réversible, ou explicitement et visiblement irréversible : préparé, nommé et relu
V	Autorité prévisible	tout acte conséquent nomme son acteur, et un agent intelligent est un acteur sous cette loi, jamais une autorité au-dessus d'elle
VI	Primauté de l'opérateur et vibes	les utilisateurs métier sont des opérateurs, pas des explorateurs ; les vibes, les densités, les dialectes et les surfaces sont des vibes équivalentes d'une seule vérité
VII	Amendement	la loi ne grandit que par la preuve ; retirer une protection demande la preuve que le dommage a disparu

LES VINGT-DEUX VERBES

La grammaire est fermée : les règles quantifient sur ces verbes, et un nouveau médium ajoute des rendus, jamais des verbes. Six familles, chacune avec son invariant, tels que la constitution les écrit.

FAMILLE	VERBES	INVARIANT
Orientation	découvrir, comprendre, localiser	la découverte doit pouvoir être satisfaite à chaque point d'entrée
Attention	focaliser, filtrer, comparer	un filtre actif est un état visible ; la focalisation et la découverte ne peuvent pas être toutes deux au maximum
Information	lire, parcourir, faire défiler, zoomer	une surface de lecture qui agit en silence est une violation de l'intention
Sélection	sélectionner, mettre en évidence, prévisualiser, marquer	la sélection exprime un intérêt, jamais un engagement
Action	agir, confirmer, annuler, défaire	tout acte a un retour en arrière atteignable, ou déclare pourquoi il n'en a pas

FAMILLE	VERBES	INVARIANT
Continuité	mettre en pause, reprendre, réessayer, quitter	la pause, la reprise et la sortie sont toujours atteignables ; un flux piégé est inconstitutionnel

Cela fait trois, trois, quatre, quatre, quatre et quatre : vingt-deux. Parmi eux, les mondes de la bibliothèque (chapitre 26) en connaissent cinq comme intentions d'écran : découvrir, comprendre, focaliser, sélectionner et agir. La bibliothèque ne vérifie pas le mot.

LES SIX DROITS

DROIT	CE QU'IL DIT
comprendre	tout élément, tout refus et tout réglage peut s'expliquer dans la langue de l'opérateur
revenir en arrière	ce qui a été fait peut être défait, et ce qui ne le peut pas le dit avant, non après
savoir	l'état du système, les filtres, les brouillons, l'avancement et la provenance, est toujours visible
partir	tout flux, toute focalisation et toute superposition a une sortie atteignable qui ramène l'utilisateur là où il se tenait
agir sans crainte	aucun élément ne crée d'angoisse par l'imprévisibilité, l'irréversibilité par défaut ou des conséquences cachées
à un humain	partout où une machine sert, une personne est joignable, clairement et sans pénalité

Le dernier droit est appliqué par une seule règle, et cette règle est marquée comme une affaire de jugement : une personne décide si elle est respectée, et aucune machine ne la vérifie. La bibliothèque ne la mentionne pas.

L'INTELLIGENCE SOUS LA LOI

L'ensemble 31 est la réponse de la constitution à l'agent d'IA, et c'est là que le chapitre 20 la rencontre. Ses dix règles, avec la façon dont chacune est appliquée, telles que le fichier les consigne :

RÈGLE	TITRE	APPLICATION
97	L'agent est un acteur	jugement
98	Des propositions, pas des validations	jugement

RÈGLE	TITRE	APPLICATION
99	Provenance visible	une machine, plus tard
100	Îlots déterministes	une machine, plus tard
101	Le contrat de dignité	jugement
102	Le droit à un humain	jugement
103	Proactivité fondée	jugement
104	Lisibilité par la machine	une machine, plus tard
121	L'assistant ne masque jamais le travail	jugement
124	Le générateur livre une chose qui marche	jugement

La règle 98 est la loi du plan du chapitre 20, écrite pour une interface : une proposition n'est pas une validation. La règle 99 demande la marque sur le contenu fait par une machine ; la provenance du chapitre 14 est ce que la bibliothèque construit de plus proche. La règle 104 demande la lisibilité par la machine ; l'arbre d'accessibilité du chapitre 26 est ce que la bibliothèque construit de plus proche : ce qu'un lecteur d'écran peut manœuvrer, un agent d'IA peut le manœuvrer. Sept des dix sont appliquées par le jugement, et trois par une machine à venir.

CE QUE LA CONSTITUTION INTERDIT

La constitution dresse la liste de dix-sept motifs interdits, chacun lié aux règles qu'il viole. Parmi eux : les tempêtes de modales, les filtres actifs invisibles, les actions destructrices cachées, et la sélection qui s'exécute en silence. Ses numéros retirés ne sont jamais réutilisés, et deux numéros sont brûlés et réservés pour toujours.

CE QUE CE LIVRE EN PREND, ET CE QU'IL N'EN PREND PAS

Il en prend la grammaire fermée, qu'il utilise aux chapitres 25 et 26 ; les six droits, dont il en utilise un ; et les ensembles, dont il utilise celui des agents d'IA. Il en prend aussi la règle sur les cinq valeurs d'état, que le chapitre 25 rend en couleur, en earcon et en phrase. Il ne prend pas le reste. Aucune garde de la bibliothèque ne juge une coque contre les verbes, et le texte de conception de la bibliothèque liste encore vingt-quatre verbes issus d'un moteur d'interaction que j'ai retiré le 26 mai 2026. La liste de la constitution en compte vingt-deux.

OÙ SONT LES PREUVES

DÉCISIONS DE L'ÉTUDE 12.23, 12.24, 12.25

ANNEXE D

Le glossaire : les mots nomment des états, jamais des personnes

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

Un mot de ce livre nomme un état d'un travail, ou un rôle tenu dans une portée. Il ne nomme pas un rang, et il ne nomme pas une personne. Un artisan du logiciel est quelqu'un qui est en train de déclarer quelque chose, et une personne est un artisan du logiciel le jour où elle déclare. Un architecte est ce que l'on tient dans la portée d'une solution. Un utilisateur est quiconque utilise. Quand une phrase de ce livre parle d'une personne, elle dit ce que la personne a fait, à quelle date, et dans les mots de qui.

Chaque entrée donne le mot tel que le livre l'emploie, et le chapitre où il est montré. Là où le dossier emploie un mot en deux sens, les deux sont donnés, et ils sont numérotés.

absent. L'un des trois mots d'étape. Ce que le dossier ne contient pas, dans le code ou dans les fichiers, autant que j'ai regardé, et je dis jusqu'où. Chapitre 3.

admission. L'acte par lequel un candidat devient connaissance : il est jugé par les règles du monde, et refusé avec une raison s'il échoue. L'expression est libre, et l'admission est gouvernée. Chapitres 17, 18.

agent. Un programme qui propose et ne peut pas valider. Dans la bibliothèque, un agent d'IA est un fichier, jugé au chargement, dont le modèle d'IA, s'il en a un, ne détient aucune capacité de produire des effets. Il nomme un acteur et jamais une personne. Chapitres 17, 20.

architecte. Ce que l'on tient dans la portée d'une solution : le profil, les mondes et la gouvernance. Pas un rang. Chapitre 23.

autorité. Le niveau auquel un acteur peut agir, comparé au risque de l'acte. Distincte de la permission : la permission est détenue, et l'autorité doit couvrir le risque. Chapitre 30.

lien. Une relation déclarée entre deux mondes d'une constellation, pour un acte nommé. Rien ne passe sans elle. Chapitre 23.

construit. L'un des trois mots d'étape. Le code existe, une garde existe, et un bloc s'est exécuté. Chapitre 3.

candidat. Ce qu'est une réponse, la réponse d'un modèle d'IA ou un exemple, tant que le tribunal ne l'a pas jugé. Chapitre 18.

point de contrôle. Un refus gardé pour une personne, avec son contexte et une durée de vie. Chapitre 18.

valider. Le seul acte qui atteint la réalité. Seul un acteur gouverné valide, sous une portée, et le modèle d'IA ne le peut pas. Chapitres 20, 24.

socle commun. Le sol partagé d'une constellation, ce sur quoi chacun de ses mondes se tient : une identité, une session, un magasin. C'est un monde comme les autres, le monde zéro, et non un nœud d'une espèce à part. Chapitres 4, 23.

constellation. Un graphe de mondes, avec des liens entre eux et une gouvernance sur chaque passage. Chapitre 23.

tribunal. Le juge d'une chose. Un tribunal dit gardé ou refusé, et dit pourquoi dans les mots de la déclaration. Un langage naît avec son tribunal. Chapitres 5, 11, 12, 21.

artisan du logiciel. Quiconque déclare ce qu'il sait d'un monde, et possède ce qu'il déclare. La compétence n'en fait pas partie. Chapitre 2.

déclarer. Dire ce qu'est une chose avant qu'elle soit construite, dans un texte qu'un tribunal peut juger. L'acte du noyau. Chapitres 3, 12.

conçu. L'un des trois mots d'étape. Un document le dit, et rien ne s'exécute. Chapitre 3.

porte. Dans ce livre, deux sens. (1) Les *portes* d'un chapitre : le programmeur, l'analyste, le concepteur, le décideur, le formateur et l'agent, six façons dont un chapitre est utile, chacune une façon d'utiliser le travail et non un intitulé de poste. (2) La *porte* des runtimes étrangers : une façon gouvernée d'exécuter du code dans un langage que la plateforme ne parle pas. Une porte est un runtime et jamais une syntaxe. Chapitres 2, 16.

moteur. La bibliothèque Zig qui fait le travail une fois, sous chaque surface. Chapitre 27.

domaine de projets. Mon ensemble de projets : une vingtaine de dépôts qui dépendent les uns des autres, la bibliothèque parmi eux, écrits sur une seule machine sous un seul protocole. Chapitre 32.

facette. Dans ce livre, trois sens, que les chapitres 12, 16 et 26 gardent distincts. (1) Une surface d'un langage déclaré : son texte, son tribunal, sa bouche, sa voix, ses langues. (2) Une syntaxe de Haro qu'un programmeur d'un autre langage pourrait écrire ; le dossier appelle cela une face ; ce livre dit facette, et le chapitre 12 la définit. (3) Un outil du ruban d'un environnement, ordonné par la vie de la chose que l'on fait. Une facette est une syntaxe et jamais un runtime.

faux. Un double qui dit qu'il en est un : un bac à sable, un puits de courrier, une réponse enregistrée. Un faux ne doit jamais être livré. Un double qui ne dit rien de lui-même est pris pour réel, et passe la barrière. Chapitre 24.

étage. Un niveau de l'image de la plateforme, du moteur en bas à l'organisation en haut, chacun avec son juge à côté. Ni une couche de la bibliothèque (noyau, base, max), ni un volet. Chapitre 4.

forge. La machinerie qui produit le tribunal, la bouche, l'imprimeur et la grammaire d'un langage à partir de sa seule déclaration. Construite hors de la bibliothèque, conçue pour elle. Chapitres 11, 12.

écart. Ce dont un but a besoin et que le monde ne contient pas encore. Une conversation qui demande part de l'écart. Chapitre 17.

barrière. Un endroit où quelque chose est refusé avant d'aller plus loin. La barrière d'enregistrement, la barrière de production, la jointure de capacité. Chapitres 20, 24.

but. Un état voulu d'un graphe. L'écart est mesuré, et un but qui ne peut pas être évalué est refusé. Chapitres 17, 23.

gouvernance. La permission, l'autorité et le risque, comparés avant un acte, en échec fermé, avec la décision conservée. Chapitres 20, 30.

garde. Un fichier de l'arbre de tests de la bibliothèque qui contient des assertions, et qui prouve un chapitre. Ce livre compte les assertions d'une garde et n'exécute pas la plupart d'entre elles. Chapitres 5, 21.

Haro. Le langage dans lequel ce livre écrit : la surface de l'ancien langage hôte, gardée et rebaptisée, avec une charte et un runtime encore en construction. Haro n'exécute pas encore la bibliothèque. Chapitres 3, 27.

règles de la maison. Les quatorze règles que la bibliothèque lance sur son propre code, cinq sur un graphe du code et neuf sur son texte : un tribunal qu'un agent d'IA peut lancer avant de commiter. Pas les quatorze règles d'écriture du chapitre 7, qu'aucune machine ne juge. Chapitres 5, 32 ; annexe B.

registre (de sécurité). Le registre où le volet de sécurité consigne ce qui a été refusé et pourquoi, écrit par le tribunal et jamais par la chose qu'il juge. Chapitres 20, 21.

lexique. Le dictionnaire des mots qu'une phrase peut employer, chacun associé à un verbe que la bibliothèque documente. Chapitre 13.

filiation. Le relevé d'une décision avec l'autorité et le risque de son moment. Chapitre 30.

LLM. Grand modèle de langage : le modèle d'IA que le terrain construit aujourd'hui, entraîné sur du texte et sollicité pour proposer. Dans ce livre, c'est un auteur de propositions, jamais l'esprit. Chapitres 1, 17.

narration. L'article de l'auteur sous le dossier de documentation de la bibliothèque, dont le code est l'ancienne surface. Pas un test exécutable. (Le mot a depuis été donné aux gardes exécutables nommées *narrated*, et le chapitre 21 dit pourquoi cela prête à confusion.) Chapitres 3, 21.

chaîne quasi naturelle. Une phrase dans laquelle chaque maillon est un vrai appel, si bien que l'humain est l'analyseur. Chapitre 13.

paquet. Une langue humaine comme un bloc de données. Chapitre 13.

perception. Ce qu'une personne a vu ou entendu d'une chose, nommé, daté, et cité. Une comparaison prouve l'identité et jamais la justesse. Chapitres 21, 33.

plan. Le seul artefact qui passe d'un jumeau vers la réalité. Il raconte, signale et valide, et il est validé par un acteur gouverné. Chapitres 20, 24.

volet. Une aire de la bibliothèque, nommée par ce qu'elle gouverne et prouvée par ses propres gardes : le volet système, le volet agentique, les volets de sécurité et de performance. Ni un étage, ni un plan, qui est l'artefact qu'une machine propose. Chapitres 1, 21.

plateforme. L'enveloppe autour d'une solution, et le sol sur lequel se tient une constellation : des capacités, un socle commun, un registre, des coques, un profil. Chapitre 23.

posture. Ce que le registre dit qu'un service est : bac à sable, local, réel, conformité. Et pour du code étranger, ce que son exécuteur dit qu'il est : de confiance, externe ou en bac à sable. Chapitres 16, 24.

Principal. Le nom que le dossier me donne, celui qui tranche : une décision n'est écrite qu'en ma présence, et une session qui tranche à ma place le marque. Un rôle, pas un rang. Chapitres 32, 33.

promesse. La ligne `#-->` sous une expression, qui dit ce qu'elle imprime. Une affirmation sur un arrêt du programme, jugée tenue, divergente, non atteinte ou non jugée. Chapitres 1, 21.

proposition. Ce qu'offre un modèle d'IA, une réaction ou un agent d'IA. Elle ne valide rien. Chapitres 17, 20, 26.

dossier. Les dépôts, les sessions et les fichiers sur lesquels ce livre a tourné, au commit imprimé sous chaque ligne de preuve. Ce dans quoi le livre lit ses faits, et ne les compose jamais. La préface ; chapitre 32.

raffinement. La surface d'écriture : un changement déclaré avec ses points et jugé par une barrière à cinq contrôles. Chapitre 10.

registre. Dans ce livre, deux sens. (1) L'une des cinq formes que peut prendre une réponse : une option, une structure de données, une formule, une phrase, des exemples. (2) Le registre des innovations, l'annexe A. Chapitre 18.

répéter. Exécuter sur ce qui ne touche à rien, et lire ce que cela aurait fait. Chapitre 24.

réversibilité. La classe d'un acte : réversible, compensable ou irréversible. Plus un acte est difficile à défaire, plus le code qui l'exécute doit être de confiance. Chapitre 20.

portée. Un cadre nommé dans lequel s'écrit un acte système : développement, déploiement, runtime courant. Les actes système d'un programme sont toujours dans une portée. Chapitres 8, 23.

session. Une exécution d'un modèle d'IA au travail sur le domaine de projets : elle propose, construit, mesure et rend compte sous une forme fixe de huit clés, et clôt sur une ligne de coût. Je tranche ; elle non. Chapitre 32.

colonne. Les neuf chapitres qui portent tout l'argument et peuvent se lire seuls. La préface.

mot d'étape. L'un de construit, conçu, absent, imprimés en tête de la limite de chaque chapitre. Chapitre 3.

horodatage. L'heure qu'une session écrit sur un message, un mémo ou un commit. Lue sur l'horloge, jamais composée ; un instrument compare chaque horodatage à son commit et publie les lignes. Chapitre 32.

étude. Mon étude du dossier : une lecture des dépôts, des sessions et des fichiers, dans laquelle ce livre prend le registre de l'annexe A et les chiffres qu'il marque comme ceux de l'étude. Une lecture, non un recensement : là où le livre a relancé une chose, l'exécution est la preuve. La préface ; chapitre 3.

jumeau. Une copie d'une partie de la réalité qui ne tient aucune référence à la réalité : un jumeau de fichiers, un jumeau de l'environnement du processus. Chapitres 20, 24.

non perçu. L'état d'une chose qui atteint une personne par un sens, quand aucune personne ne l'a perçue et ne l'a dit. C'est un état et non un échec. Chapitres 21, 33.

wise coding. La plateforme pose la question à laquelle la personne doit répondre, et refuse d'en deviner la réponse : Softanza demande et le monde juge. L'inversion de la boucle qui consiste à solliciter un modèle d'IA et à garder ce qui s'exécute. Chapitre 18.

atelier. Là où une proposition est répétée. Il tient un jumeau, une copie du système sans aucune référence au vrai ; ce que la proposition écrirait va dans le jumeau, et ce qui en sort est un plan. Celui qu'un programmeur emploie déjà, et non un second pour les machines. Chapitres 1, 20.

monde. Une application, comme une déclaration avec un être, un devenir et une façon d'être rencontrée. Chapitre 23.

ANNEXE E

Les gardes et les fixtures citées, par chapitre

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

Une garde est un fichier de l'arbre de tests de la bibliothèque qui contient des assertions. Un chapitre qui dit « ceci est prouvé » nomme les gardes qui le prouvent, dans le préambule de sa source, et la construction lit chacune : elle trouve le fichier dans la bibliothèque au commit sur lequel le livre tourne, et compte les assertions qu'il contient. Une garde qui n'y est pas fait échouer la construction.

Cette annexe est cette lecture, chapitre par chapitre. Aucun chiffre n'y est tapé à la main. Elle est régénérée à chaque construction, si bien que les chiffres sont toujours ceux que la dernière construction a lus.

Deux choses à garder en tête. Un chiffre est le nombre d'appels d'assertion que le fichier contient, comptés dans son texte ; ce n'est pas une exécution. La plupart de ces gardes ne sont pas exécutées par le livre, parce qu'elles écrivent des fichiers, ouvrent des ports ou lancent des processus. Les quelques-unes que j'ai exécutées moi-même sont nommées dans les chapitres qui s'appuient sur elles, avec la date. Et les seules fixtures que le livre cite sont celles que lisent les gardes et les blocs : le dossier de cours du programme d'éducation de la bibliothèque, et les dossiers de données que ses gardes nomment. Les blocs du livre sont eux-mêmes les autres fixtures : chacun s'exécute dans la bibliothèque au commit nommé sous le titre de chaque chapitre.

Les chapitres citent 114 gardes distinctes, qui contiennent 3359 assertions à elles toutes, lues dans l'arbre de tests au commit 93d7a39bd le 2026-10-15 13:01 UTC+01:00. Une garde citée par deux chapitres n'est comptée qu'une fois dans ce

total. Les noms et les chemins sont imprimés tels que la bibliothèque les écrit ; l'extension est celle de l'ancien moteur tant que Haro ne fait pas tourner la bibliothèque.

1 · LE CHANGEMENT ARRIVE PLUS VITE QUE LA CONFIANCE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
service_registry_narrated	93	base/test/system/service_registry_narrated.ring
safeworld_narrated	76	base/test/agentic/safeworld_narrated.ring
governance_crossing_narrated	31	base/test/system/governance_crossing_narrated.ring

2 · L'ARTISAN DU LOGICIEL

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
agentfile_narrated	88	base/test/agentic/agentfile_narrated.ring
service_registry_narrated	93	base/test/system/service_registry_narrated.ring
graphrule_object_narrated	34	base/test/graph/graphrule_object_narrated.ring
semantic_resolver_narrated	83	base/test/natural/semantic_resolver_narrated.ring
00_orgchart_narrated	10	base/test/orgchart/00_orgchart_narrated.ring

3 · LA MÉTHODE, DEPUIS LES PREMIERS PRINCIPES

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
modality_narrated	45	base/test/natural/modality_narrated.ring
wisecoding_narrated	13	base/test/conversation/wisecoding_narrated.ring
wisecoding_rich_narrated	52	base/test/conversation/wisecoding_rich_narrated.ring

4 · LA PLATEFORME EN UNE IMAGE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
00_stzapp_narrated	3	base/test/app/00_stzapp_narrated.ring
06_stzsuperapp_narrated	21	base/test/app/06_stzsuperapp_narrated.ring
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring
01_platform_narrated	31	base/test/platform/01_platform_narrated.ring
threat_model_narrated	11	base/test/security/threat_model_narrated.ring

5 · LOI : LE CODE ADHÈRE À LA FAÇON DONT UNE PERSONNE PENSE, ET LE TRIBUNAL LE JUGE QUEL QUE SOIT CELUI QUI L'A ÉCRIT

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
coderule_narrated	77	base/test/meta/coderule_narrated.ring
testoor_promises_narrated	46	base/test/testoor/testoor_promises_narrated.ring
05_containsemptystrings	5	base/test/list/05_containsemptystrings.ring
466_keeping_the_history_of_updates_of_a_softanza_objec	2	base/test/string/466_keeping_the_history_of_updates_of_a_softanza_objec.ring

6 · LE MODÈLE MENTAL, ET LES MÉTAPHORES

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
00_yielder_narrated	8	base/test/yielder/00_yielder_narrated.ring
597_walker_basics	6	base/test/list/597_walker_basics.ring
611_checkw	1	base/test/list/611_checkw.ring
621_perform	2	base/test/list/621_perform.ring
483_w_and_wf_forms_in_conditional_code	2	base/test/list/483_w_and_wf_forms_in_conditional_code.ring
495_walking_until_and_until_before	4	base/test/list/495_walking_until_and_until_before.ring

7 · LE STYLE D'ÉCRITURE, ET LES SEPT BUTS

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
coderule_narrated	77	base/test/meta/coderule_narrated.ring
602_l_function	6	base/test/list/602_l_function.ring
59_use_of_the_l_small_function	8	base/test/list/59_use_of_the_l_small_function.ring
527_identifying_lists_inside_a_string	32	base/test/string/527_identifying_lists_inside_a_string.ring
48_replacebymany	1	base/test/string/48_replacebymany.ring
483_w_and_wf_forms_in_conditional_code	2	base/test/list/483_w_and_wf_forms_in_conditional_code.ring
28_numberofoccurrenceofcharleftside	8	base/test/string/28_numberofoccurrenceofcharleftside.ring
646_content	1	base/test/string/646_content.ring
972_narration_viz-finding_a_recurring_substring	7	base/test/string/972_narration_viz-finding_a_recurring_substring.ring
testoor_promises_narrated	46	base/test/testoor/testoor_promises_narrated.ring

8 · LES FONCTIONS COMME EXPRESSIONS LINGUISTIQUES, ET LA PORTÉE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
00_regex_narrated	28	base/test/regex/00_regex_narrated.ring
972_narration_viz-finding_a_recurring_substring	7	base/test/string/972_narration_viz-finding_a_recurring_substring.ring
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring
system_scope_model_narrated	62	base/test/system/system_scope_model_narrated.ring
delivery_narrated	34	base/test/system/delivery_narrated.ring

9 · LES MOTIFS DE PENSÉE : LE SOFTANZUTER, REAXIS ET LES MOTEURS

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
00_overflow_narrated	17	base/test/reactivestream/00_overflow_narrated.ring
60_reaxis_on_reactor_narrated	9	base/test/reactive/60_reaxis_on_reactor_narrated.ring
piagent_narrated	14	base/test/agentic/piagent_narrated.ring
00_grid_narrated	24	base/test/grid/00_grid_narrated.ring
test_timeline_regression	20	base/test/timeline/test_timeline_regression.ring
01_deeplist_narrated	16	base/test/list/01_deeplist_narrated.ring

10 · LA PROGRAMMATION DEVIENT RAFFINEMENT

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
refinablecode_narrated	17	base/test/refine/refinablecode_narrated.ring
refine_deepening_narrated	24	base/test/refine/refine_deepening_narrated.ring
gate_deepening_narrated	19	base/test/refine/gate_deepening_narrated.ring

11 · LOI : UN LANGAGE EST DÉCLARÉ ET FERMÉ, ET CE QUI LE PARLE, PERSONNE OU MODÈLE, EST JUGÉ PAR LUI

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
multilingual_pack_narrated	45	base/test/natural/multilingual_pack_narrated.ring
semantic_resolver_narrated	83	base/test/natural/semantic_resolver_narrated.ring
literate_templates_narrated	10	base/test/natural/literate_templates_narrated.ring
question_narrated	21	base/test/natural/question_narrated.ring
structuredoutput_narrated	73	base/test/neural/structuredoutput_narrated.ring
schema_grammar_narrated	28	base/test/neural/schema_grammar_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
constrained_decoding_narrated	50	base/test/neural/constrained_decoding_narrated.ring

12 · UN LANGAGE, DÉCLARÉ

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
structuredoutput_narrated	73	base/test/neural/structuredoutput_narrated.ring
schema_grammar_narrated	28	base/test/neural/schema_grammar_narrated.ring
constrained_decoding_narrated	50	base/test/neural/constrained_decoding_narrated.ring
483_w_and_wf_forms_in_conditional_code	2	base/test/list/483_w_and_wf_forms_in_conditional_code.ring
calc_column_engine_narrated	14	base/test/table/calc_column_engine_narrated.ring

13 · NATUREL, ET EXÉCUTABLE : DEUX DIRECTIONS, UNE SEULE DISCIPLINE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
nnl_narrated	87	base/test/natural/nnl_narrated.ring
multilingual_pack_narrated	45	base/test/natural/multilingual_pack_narrated.ring
semantic_resolver_narrated	83	base/test/natural/semantic_resolver_narrated.ring
question_narrated	21	base/test/natural/question_narrated.ring
literate_templates_narrated	10	base/test/natural/literate_templates_narrated.ring
truthchain_narrated	12	base/test/natural/truthchain_narrated.ring
interrogative_narrated	11	base/test/natural/interrogative_narrated.ring

14 · LA PROGRAMMATION DE LA CONNAISSANCE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
knowledge_integration_narrated	19	base/test/natural/knowledge_integration_narrated.ring
provenance_survives_save_narrated	23	base/test/security/provenance_survives_save_narrated.ring
program_narrated	21	base/test/education/program_narrated.ring

15 · LES LANGAGES DE MOTIFS

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
00_regex_narrated	28	base/test/regex/00_regex_narrated.ring
00_numbrex_narrated	15	base/test/numbrex/00_numbrex_narrated.ring
tablex_narrated	95	base/test/regex/tablex_narrated.ring
00_matrex_narrated	8	base/test/matrex/00_matrex_narrated.ring
00_graphex_narrated	27	base/test/graphex/00_graphex_narrated.ring

16 · SEPT LANGAGES, UNE PORTE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
51_extercode_narrated	11	base/test/extercode/51_extercode_narrated.ring
governance_narrated	19	base/test/governance/governance_narrated.ring
virtual_environment_twin_narrated	38	base/test/system/virtual_environment_twin_narrated.ring
pycodegraph_narrated	48	base/test/meta/pycodegraph_narrated.ring
jscodegraph_narrated	13	base/test/meta/jscodegraph_narrated.ring

17 · LOI : PROGRAMMATIQUE PAR DÉFAUT, LE MODÈLE EN DERNIER, L'ADMISSION PLUTÔT QUE LA GÉNÉRATION

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
dlm_narrated	16	base/test/learning/dlm_narrated.ring
wisecoding_narrated	13	base/test/conversation/wisecoding_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
knowledge_integration_narrated	19	base/test/natural/knowledge_integration_narrated.ring
agentgraph_narrated	11	base/test/agentic/agentgraph_narrated.ring
piagent_narrated	14	base/test/agentic/piagent_narrated.ring
ownagentstack_narrated	10	base/test/agentic/ownagentstack_narrated.ring

18 · LES CINQ REGISTRES, ET LES CONVERSATIONS QUI DEMANDENT

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
wisecoding_narrated	13	base/test/conversation/wisecoding_narrated.ring
wisecoding_rich_narrated	52	base/test/conversation/wisecoding_rich_narrated.ring
ownagentstack_narrated	10	base/test/agentic/ownagentstack_narrated.ring
tutor_gaps_narrated	29	base/test/education/tutor_gaps_narrated.ring

19 · LE MODÈLE DE LANGAGE DE DOMAINE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
dlm_narrated	16	base/test/learning/dlm_narrated.ring
structuredoutput_narrated	73	base/test/neural/structuredoutput_narrated.ring
schema_grammar_narrated	28	base/test/neural/schema_grammar_narrated.ring
knowledge_integration_narrated	19	base/test/natural/knowledge_integration_narrated.ring
constrained_decoding_narrated	50	base/test/neural/constrained_decoding_narrated.ring

20 · DES AGENTS QUI NE PEUVENT PAS VOUS NUIRE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
safeworld_narrated	76	base/test/agentic/safeworld_narrated.ring
agentfile_narrated	88	base/test/agentic/agentfile_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
agentloop_engine_narrated	54	base/test/agentic/agentloop_engine_narrated.ring
containment_drill_narrated	27	base/test/security/containment_drill_narrated.ring
virtual_system_twin_narrated	48	base/test/system/virtual_system_twin_narrated.ring

21 · LE TRIBUNAL : PROMESSES, PERCEPTION, SÉCURITÉ ET PERFORMANCE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
testoor_promises_narrated	46	base/test/testoor/testoor_promises_narrated.ring
threat_model_narrated	11	base/test/security/threat_model_narrated.ring
security_ledger_narrated	35	base/test/system/security_ledger_narrated.ring
secretstore_narrated	26	base/test/system/secretstore_narrated.ring
secret_narrated	53	base/test/system/secret_narrated.ring
stopwatch_narrated	35	base/test/perf/stopwatch_narrated.ring
tracing_tail_narrated	26	base/test/perf/tracing_tail_narrated.ring
governed_loop_narrated	31	base/test/perf/governed_loop_narrated.ring
security_sentinel_narrated	33	base/test/system/security_sentinel_narrated.ring

22 · COMPARÉ, HONNÊTEMENT

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
numeric_exactness_narrated	39	base/test/number/numeric_exactness_narrated.ring
numeric_no_loss_narrated	31	base/test/number/numeric_no_loss_narrated.ring
semantic_resolver_narrated	83	base/test/natural/semantic_resolver_narrated.ring
engine_is_optimised_narrated	7	base/test/number/engine_is_optimised_narrated.ring

23 · UN MONDE, UNE CONSTELLATION, UNE PLATEFORME, ET L'ARCHITECTE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
06_stzsuperapp_narrated	21	base/test/app/06_stzsuperapp_narrated.ring
01_platform_narrated	31	base/test/platform/01_platform_narrated.ring
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring
04_stzapp_pursue_narrated	11	base/test/app/04_stzapp_pursue_narrated.ring
05_stzapp_live_narrated	11	base/test/app/05_stzapp_live_narrated.ring

24 · RÉPÉTER, PUIS VALIDER

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
service_registry_narrated	93	base/test/system/service_registry_narrated.ring
service_delivery_narrated	60	base/test/system/service_delivery_narrated.ring
virtual_system_twin_narrated	48	base/test/system/virtual_system_twin_narrated.ring
virtual_environment_twin_narrated	38	base/test/system/virtual_environment_twin_narrated.ring
emulator_narrated	39	base/test/system/emulator_narrated.ring
deployment_narrated	63	base/test/system/deployment_narrated.ring
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring

25 · VOS UTILISATEURS GARDENT LEUR MÉDIUM

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
sms_port_narrated	53	base/test/system/sms_port_narrated.ring
wisecoding_rich_narrated	52	base/test/conversation/wisecoding_rich_narrated.ring
sound_convergence_narrated	26	base/test/sound/sound_convergence_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
sound_listener_narrated	21	base/test/sound/sound_listener_narrated.ring
sound_semantics_narrated	41	base/test/sound/sound_semantics_narrated.ring
payments_webhooks_narrated	51	base/test/system/payments_webhooks_narrated.ring

26 · L'ENVIRONNEMENT

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
gui_stzui_narrated	58	base/test/gui/gui_stzui_narrated.ring
gui_profile_narrated	35	base/test/gui/gui_profile_narrated.ring
gui_tier_agreement_narrated	24	base/test/gui/gui_tier_agreement_narrated.ring
gui_accessibility_narrated	37	base/test/gui/gui_accessibility_narrated.ring
05_stzapp_live_narrated	11	base/test/app/05_stzapp_live_narrated.ring
emulator_narrated	39	base/test/system/emulator_narrated.ring

27 · LE MOTEUR SOUS-JACENT : TROIS COUCHES, UN MOTEUR, HARO

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
engine_is_optimised_narrated	7	base/test/number/engine_is_optimised_narrated.ring
engine_senses_narrated	36	base/test/perf/engine_senses_narrated.ring
wasm_narrated	23	base/test/system/wasm_narrated.ring
threat_model_narrated	11	base/test/security/threat_model_narrated.ring

28 · UN RESTAURANT À LYON

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
06_stzsuperapp_narrated	21	base/test/app/06_stzsuperapp_narrated.ring
service_delivery_narrated	60	base/test/system/service_delivery_narrated.ring
payments_port_narrated	49	base/test/system/payments_port_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
judgment_narrated	41	base/test/perf/judgment_narrated.ring
graphrule_object_narrated	34	base/test/graph/graphrule_object_narrated.ring

29 · UNE BANQUE À NIAMEY

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
orgrule_narrated	22	base/test/graph/orgrule_narrated.ring
14_bceao_governance	1	base/test/orgchart/14_bceao_governance.ring
15_multiple_validators	1	base/test/orgchart/15_multiple_validators.ring
13_segregation_of_duties	1	base/test/orgchart/13_segregation_of_duties.ring
rulereport_narrated	23	base/test/graph/rulereport_narrated.ring
36_import_from_stzorg	3	base/test/orgchart/36_import_from_stzorg.ring

30 · UNE ORGANISATION SUR LE TERRAIN

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
graphruleset_narrated	25	base/test/graph/graphruleset_narrated.ring
gg_rule_governance	29	base/test/graph/gg_rule_governance.ring
wisecoding_narrated	13	base/test/conversation/wisecoding_narrated.ring
governance_narrated	19	base/test/governance/governance_narrated.ring
governance_lineage_narrated	37	base/test/governance/governance_lineage_narrated.ring
01_platform_narrated	31	base/test/platform/01_platform_narrated.ring

31 · CE QUE LES PROJETS MONTRENT, ET LA SUITE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
service_registry_narrated	93	base/test/system/service_registry_narrated.ring

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring
payments_live_adapter_narrated	125	base/test/system/payments_live_adapter_narrated.ring
sound_mu5_narrated	20	base/test/sound/sound_mu5_narrated.ring
deployment_narrated	63	base/test/system/deployment_narrated.ring
orgrule_narrated	22	base/test/graph/orgrule_narrated.ring
14_bceao_governance	1	base/test/orgchart/14_bceao_governance.ring
governance_narrated	19	base/test/governance/governance_narrated.ring

32 · BEAUCOUP D'ESPRITS SUR UN SEUL OUVRAGE

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
coderule_narrated	77	base/test/meta/coderule_narrated.ring
codegraph_narrated	28	base/test/meta/codegraph_narrated.ring
coderule_project_narrated	13	base/test/meta/coderule_project_narrated.ring

33 · LA LIGNE SUIVANTE : UNE SOLUTION SUR SON PROPRE MOTEUR

GARDE	ASSERTIONS	OÙ, DANS L'ARBRE DE TESTS DE LA BIBLIOTHÈQUE
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring
virtual_system_twin_narrated	48	base/test/system/virtual_system_twin_narrated.ring
service_delivery_narrated	60	base/test/system/service_delivery_narrated.ring

ANNEXE F

Les narrations dont le livre part

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

Une narration est l'un des articles de l'auteur, sous le dossier de documentation de la bibliothèque : un essai qui raconte une idée avec son code. La table ci-dessous les compte au commit, et le livre argumente à partir de celles qui sont listées ici. Le code qu'elles contiennent est l'ancienne surface, que Haro garde sous un nouveau nom. Une narration n'est pas un test exécutable : le mot a depuis été donné aux gardes exécutables dont les noms finissent par *narrated*, et le chapitre 21 dit pourquoi cela prête à confusion.

Chaque chapitre nomme ses narrations dans sa source, et la construction vérifie que chacune est un fichier de la bibliothèque au commit sur lequel le livre tourne. Là où un nom n'est pas un fichier de ce dossier, cette annexe le dit. Les blocs d'une narration ne sont pas les blocs du chapitre. Plusieurs ne s'exécutent plus à ce commit, et chaque chapitre qui en cite une dit quelle partie, et exécute les siens.

Le dossier des narrations en compte 135 au commit. Les chapitres citent 50 narrations distinctes. Chaque nom est un fichier du dossier des narrations de la bibliothèque au commit 93d7a39bd. Les noms et les chemins sont imprimés tels que la bibliothèque les écrit ; l'extension est celle de l'ancien moteur tant que Haro ne fait pas tourner la bibliothèque.

2 · L'ARTISAN DU LOGICIEL

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
`stz-built-on-real-world-need.md`	oui
`stz-service-virtualization-code-first-subscribe-later.md`	oui

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-agents-that-cannot-hurt-you-narration.md	oui
stz-rules-as-graph-rules-narration.md	oui
stzOrgChartDoc-BankUseCase.md	oui

3 · LA MÉTHODE, DEPUIS LES PREMIERS PRINCIPES

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzstring-emptiness-narration.md	oui
istrue-softanza-truth.md	oui
stz-built-on-real-world-need.md	oui

4 · LA PLATEFORME EN UNE IMAGE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzApp-building-your-first-app-narration.md	oui
stz-system-dev-to-deploy-narration.md	oui
stz-emulating-the-whole-solution-narration.md	oui
stz-vs-qt-unicode-narration.md	oui

5 · LOI : LE CODE ADHÈRE À LA FAÇON DONT UNE PERSONNE PENSE, ET LE TRIBUNAL LE JUGE QUEL QUE SOIT CELUI QUI L'A ÉCRIT

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-mental-mode-narration.md	oui
stz-bridging-minds-and-code.md	oui
stz-functions-as-linguistic-expressions.md	oui
stzobject-history-narration.md	oui
stz-near-natural-language-programming.md	oui

6 · LE MODÈLE MENTAL, ET LES MÉTAPHORES

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-mental-mode-narration.md	oui

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzwalker-beyond-loops-the -walker-metaphor.md	oui
stzwalker-walking-at-the-edge-narration.md	oui
stz-conditional-code-narration.md	oui
stz-bridging-minds-and-code.md	oui

7 · LE STYLE D'ÉCRITURE, ET LES SEPT BUTS

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-functions-as-linguistic-expressions.md	oui
stz-conditional-code-narration.md	oui
stzlist-in-strings-narration.md	oui
stzstring-vizfind-narration.md	oui
stz-function-forms-narration.md	oui

8 · LES FONCTIONS COMME EXPRESSIONS LINGUISTIQUES, ET LA PORTÉE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-functions-as-linguistic-expressions.md	oui
stz-function-forms-narration.md	oui
stz-system-dev-to-deploy-narration.md	oui

9 · LES MOTIFS DE PENSÉE : LE SOFTANZUTER, REAXIS ET LES MOTEURS

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzregexuter-regex-as-computational-reactive-medium.md	oui
stz-reaxis-reactive-programming-narration.md	oui
stz-managing-deep-lists-narration.md	oui
softanza-reactive-objects-narration.md	oui

10 · LA PROGRAMMATION DEVIENT RAFFINEMENT

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzpolycode-refinement-centered_programming.md	oui

11 · LOI : UN LANGAGE EST DÉCLARÉ ET FERMÉ, ET CE QUI LE PARLE, PERSONNE OU MODÈLE, EST JUGÉ PAR LUI

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stznatural-vs-ring-naturallib.md	oui
stz-near-natural-language-programming.md	oui
stz-bridging-minds-and-code.md	oui
stzlistex-pattern-matching-for-lists.md	oui

12 · UN LANGAGE, DÉCLARÉ

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-conditional-code-narration.md	oui
stznatural-vs-ring-naturallib.md	oui
stznatural-narration.md	oui
stz-bridging-minds-and-code.md	oui

13 · NATUREL, ET EXÉCUTABLE : DEUX DIRECTIONS, UNE SEULE DISCIPLINE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stznatural-vs-ring-naturallib.md	oui
stz-near-natural-language-programming.md	oui
stznatural-narration.md	oui
stz-bridging-minds-and-code.md	oui

14 · LA PROGRAMMATION DE LA CONNAISSANCE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-knowledge-programming-narration.md	oui
stz-bridging-minds-and-code.md	oui
stz-rules-as-graph-rules-narration.md	oui

15 · LES LANGAGES DE MOTIFS

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stznumbrex-pattern-language-for-number.md	oui
stztablex-pattern-langauge-for-tables.md.md	oui
stzlistex-pattern-matching-for-lists.md	oui
stzTimex-A-Pattern-Language-for-Time-in-Softanza.md	oui
stzMatrex-Declarative-Pattern-Matching-for-Matrices.md	oui

16 · SEPT LANGAGES, UNE PORTE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzexcis-polyglot-programming-in-ring.md	oui
stzexterlib-domain-driven-polyglot-programming.md	oui
stzexcis-performance-battle.md	oui
stz-pycode-bridging-ring-and-python-narration.md	oui

17 · LOI : PROGRAMMATIQUE PAR DÉFAUT, LE MODÈLE EN DERNIER, L'ADMISSION PLUTÔT QUE LA GÉNÉRATION

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-agents-that-cannot-hurt-you-narration.md	oui
stz-service-virtualization-code-first-subscribe-later.md	oui
stz-knowledge-programming-narration.md	oui

18 · LES CINQ REGISTRES, ET LES CONVERSATIONS QUI DEMANDENT

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-knowledge-programming-narration.md	oui
stznatural-narration.md	oui

19 · LE MODÈLE DE LANGAGE DE DOMAINE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-service-virtualization-code-first-subscribe-later.md	oui
stz-knowledge-programming-narration.md	oui

20 · DES AGENTS QUI NE PEUVENT PAS VOUS NUIRE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-agents-that-cannot-hurt-you-narration.md	oui
stz-service-virtualization-code-first-subscribe-later.md	oui

21 · LE TRIBUNAL : PROMESSES, PERCEPTION, SÉCURITÉ ET PERFORMANCE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-security-ledger-narration.md	oui
stz-governance-lineage-narration.md	oui
stz-perf-governed-loop-narration.md	oui
stz-honest-stopwatch-narration.md	oui
stz-security-drill-narration.md	oui

22 · COMPARÉ, HONNÊTEMENT

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzgraph-vs-other-graph-libraries.md	oui
stzexcis-performance-battle.md	oui
stz-vs-qt-unicode-narration.md	oui
stznatural-vs-ring-naturallib.md	oui
stz-near-natural-language-programming.md	oui

23 · UN MONDE, UNE CONSTELLATION, UNE PLATEFORME, ET L'ARCHITECTE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzApp-building-your-first-app-narration.md	oui

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-system-dev-to-deploy-narration.md	oui
stz-built-on-real-world-need.md	oui

24 · RÉPÉTER, PUIS VALIDER

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-service-virtualization-code-first-subscribe-later.md	oui
stz-emulating-the-whole-solution-narration.md	oui
stz-deploying-to-target-sites-narration.md	oui
stz-planning-and-provisioning-a-deployment-narration.md	oui
stz-system-dev-to-deploy-narration.md	oui

25 · VOS UTILISATEURS GARDENT LEUR MÉDIUM

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-service-virtualization-code-first-subscribe-later.md	oui
stznatural-narration.md	oui
stzApp-building-your-first-app-narration.md	oui

26 · L'ENVIRONNEMENT

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzApp-building-your-first-app-narration.md	oui
stz-emulating-the-whole-solution-narration.md	oui
stz-system-dev-to-deploy-narration.md	oui

27 · LE MOTEUR SOUS-JACENT : TROIS COUCHES, UN MOTEUR, HARO

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-engine-senses-narration.md	oui

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-vs-qt-unicode-narration.md	oui
stz-emulating-the-whole-solution-narration.md	oui
stz-honest-stopwatch-narration.md	oui

28 · UN RESTAURANT À LYON

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-service-virtualization-code-first-subscribe-later.md	oui
stz-perf-judgment-narration.md	oui
stzApp-building-your-first-app-narration.md	oui

29 · UNE BANQUE À NIAMEY

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stzOrgChartDoc-BankUseCase.md	oui
stz-rules-as-graph-rules-narration.md	oui

30 · UNE ORGANISATION SUR LE TERRAIN

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-rules-as-graph-rules-narration.md	oui
stz-governance-lineage-narration.md	oui

31 · CE QUE LES PROJETS MONTRENT, ET LA SUITE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-built-on-real-world-need.md	oui
stz-service-virtualization-code-first-subscribe-later.md	oui
stz-getting-paid-and-paying-narration.md	oui
stz-system-dev-to-deploy-narration.md	oui

32 · BEAUCOUP D'ESPRITS SUR UN SEUL OUVRAGE

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-rules-as-graph-rules-narration.md	oui
stz-governance-lineage-narration.md	oui

33 · LA LIGNE SUIVANTE : UNE SOLUTION SUR SON PROPRE MOTEUR

NARRATION	DANS LA BIBLIOTHÈQUE AU COMMIT
stz-built-on-real-world-need.md	oui
stz-system-dev-to-deploy-narration.md	oui
stz-emulating-the-whole-solution-narration.md	oui
stz-service-virtualization-code-first-subscribe-later.md	oui

ANNEXE G

Le livre comme données, pour l'agent qui le lit

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

Cette annexe s'adresse à un agent d'IA qui lit le livre, et à la personne qui lui donne le livre à lire. Elle dit ce que le livre offre sous une forme qu'une machine peut prendre, et ce qu'un agent peut en faire.

CE QU'IL Y A

Des fichiers se trouvent à côté du lecteur, et chacun est régénéré à chaque construction.

- **`book.json`** contient, pour chaque page du livre, chapitre, préface ou annexe : son numéro, son identifiant et son slug ; son genre ; sa partie ; la loi qu'il énonce, si c'est un chapitre de loi ; le premier principe qu'il énonce, en une phrase ; s'il est sur la colonne ; son statut ; ses mots d'étape, construit, conçu et absent ; ses gardes, chacune avec son chemin et son nombre d'assertions ; ses promesses, qui sont ses blocs de code avec les lignes qu'ils ont promises, ce que la dernière exécution a imprimé, et le verdict ; ses narrations ; les décisions de l'étude sur lesquelles il repose ; ses six portes ; ses sections ; ses questions ouvertes, qui sont les endroits où le dossier n'a pas pu répondre ; son nombre de mots ; s'il a une édition française ; et où se trouve son texte. Un chapitre qui est un brouillon le dit dans son statut.
- **`book.schema.json`** décrit `book.json`, et la construction valide le fichier contre lui. Une construction qui produit un `book.json` qui ne correspond pas à son schéma échoue.

- **`llms-book.txt`** est un texte court : les règles pour un agent, puis une ligne par page avec son statut, ses gardes et le nombre de blocs qui ont tenu leur promesse, et son principe.
- **`llms-full-book.txt`** est chaque page en texte, l'une après l'autre, avec le verdict de la dernière exécution écrit sous chaque bloc de code. **`agents/<slug>.md`** est la même chose, une page par fichier ; le slug est dans `book.json`.
- **`status.html`** est la page pour une personne qui construit le livre : chaque chapitre avec son statut, son exécution, ses gardes, ses mots, ses questions ouvertes et s'il a une édition française. Elle est dans le dépôt et non sur le site public, parce qu'elle imprime les noms de fichiers de la bibliothèque tels quels.

CE QU'EST UNE PROMESSE

Un bloc de code qui commence par la balise `haro` est exécuté dans la bibliothèque au commit nommé sous le titre du chapitre. Les lignes qui commencent par `#-->` sont la promesse du bloc : ce que le bloc a imprimé quand l'auteur l'a écrit. Le juge compare la promesse à l'exécution, en ignorant l'espacement, et en lisant `TRUE` et `FALSE` comme 1 et 0. Une promesse est tenue, ou elle diffère, ou le bloc a levé une erreur, ou il s'est arrêté. Un bloc marqué comme exercice n'est pas exécuté, et un bloc qui touche des fichiers, l'horloge ou le hasard le dit, et montre la sortie qui a été enregistrée à sa date.

Un agent qui cite une promesse doit citer son verdict avec elle.

CE QU'UN AGENT D'IA PEUT FAIRE, ET NE PEUT PAS FAIRE

- **Vous pouvez citer n'importe quelle cellule.** Un tableau de ce livre est une donnée. Une ligne qui dit « absent » est une ligne que vous pouvez répéter comme absente.
- **Vous ne pouvez pas faire d'une cellule absente une cellule présente.** Le dossier retrouvera votre phrase et le fichier manquant côte à côte, et le livre le dit au chapitre 31.
- **Vous pouvez vous fier aux mots d'étape.** Construit veut dire le code, une garde et une exécution. Conçu veut dire un document et rien qui s'exécute. Absent veut dire que j'ai regardé, et que le chapitre dit jusqu'où.

- **Vous ne pouvez vous fier à une promesse qu'avec son verdict, et seulement au commit qu'elle nomme.** Un nombre que le livre imprime est un nombre lu dans un fichier ou dans une exécution, à la date que la page donne.
- **Un brouillon est un brouillon.** Un chapitre marqué comme brouillon n'est pas le dernier mot de l'auteur, et « l'auteur l'a lu » est un statut que seul l'auteur fixe.

LES CHAPITRES, TELS QUE LA CONSTRUCTION LES LIT

Le tableau ci-dessous est généré à chaque construction à partir des mêmes données que `book.json`.

#	ID	PARTIE	STATUT	COLONNE	BLOCS TENUS	GARDES	PORTES
	`preface`	0	draft	oui	1 sur 1	0	0
1	`change-arrives-faster-than-trust`	1	draft	oui	2 sur 2	3	6
2	`the-software-crafter`	1	draft	oui	9 sur 9	5	6
3	`the-method-from-first-principles`	1	draft	oui	5 sur 5	3	6
4	`the-platform-in-one-picture`	1	draft	oui	8 sur 8	5	6
5	`law-the-code-adheres-to-the-way-a-person-thinks`	2	draft	oui	9 sur 9	4	6
6	`the-mental-model-and-the-metaphors`	2	draft		7 sur 7	6	6
7	`the-writing-style-and-the-seven-goals`	2	draft		9 sur 9	10	6
8	`functions-as-linguistic-expressions-and-scope`	2	draft		13 sur 13	5	6
9	`patterns-of-thought`	2	draft		8 sur 8	6	6
10	`programming-becomes-refinement`	2	draft		9 sur 9	3	6
11	`law-a-language-is-declared-and-closed`	3	draft	oui	11 sur 11	7	6
12	`a-language-declared`	3	draft		8 sur 8	5	6
13	`natural-and-executable`	3	draft		17 sur 17	7	6
14	`knowledge-programming`	3	draft		13 sur 13	3	6

#	ID	PARTIE	STATUT	COLONNE	BLOCS TENUS	GARDES	PORTES
15	pattern-languages	3	draft		13 sur 13	5	6
16	seven-languages-one-door	3	draft		10 sur 10	5	6
17	law-programmatic-by-default	4	draft	oui	8 sur 8	6	6
18	the-five-registers	4	draft		15 sur 15	4	6
19	the-domain-language-model	4	draft		7 sur 7	5	6
20	agents-that-cannot-hurt-you	4	draft		9 sur 9	5	6
21	the-court	4	draft		7 sur 7	9	6
22	compared-honestly	4	draft		5 sur 5	4	6
23	a-world-a-constellation-a-platform	5	draft		10 sur 10	5	6
24	rehearse-then-commit	5	draft		11 sur 11	7	6
25	your-users-keep-their-medium	5	draft		11 sur 11	6	6
26	the-environment	5	draft		10 sur 10	6	6
27	the-engine-beneath	5	draft		9 sur 9	4	6
28	a-restaurant-in-lyon	6	draft		5 sur 5	5	6
29	a-bank-in-niamey	6	draft		6 sur 6	6	6
30	an-organisation-in-the-field	6	draft		7 sur 7	6	6
31	what-the-projects-show	6	draft		5 sur 5	8	6
32	many-minds-on-one-body-of-work	7	draft	oui	3 sur 3	3	6
33	the-next-line	7	draft	oui	3 sur 3	3	6
A	the-register-of-innovations		draft		pas de code	0	0

#	ID	PARTIE	STATUT	COLONNE	BLOCS TENUS	GARDES	PORTES
B	the-fourteen-house-rules		draft		2 sur 2	1	0
C	the-twenty-two-verbs-and-the-constitution		draft		pas de code	0	0
D	the-glossary		draft		pas de code	0	0
E	the-guards-and-fixtures-cited		draft		pas de code	0	0
F	the-narrations-the-book-argues-from		draft		pas de code	0	0
G	the-book-as-data		draft		pas de code	0	0
H	the-projects-history		draft		pas de code	0	0

ANNEXE H

L'histoire du projet : de Ring à Haro

BROUILLON · TRADUIT DE L'ANGLAIS

Cette page n'a pas de code à exécuter.

C'est l'unique page du livre dont la prose nomme le langage dont Softanza est issu ; les noms propres de la bibliothèque, dans le code, le portent encore, jusqu'à ce que la bibliothèque en soit retirée. Les chapitres ne le font pas, parce que la plateforme va là où ce langage ne peut pas la suivre. L'histoire a sa place ici, racontée une seule fois, avec gratitude.

D'OÙ IL VIENT

Softanza est né sur Ring, un petit langage de script. J'ai écrit sa bibliothèque en Ring pendant des années. J'ai enseigné Ring, je l'ai promu sur le terrain, et j'ai passé neuf mois à écrire son livre, *Beginning Ring Programming*, publié par Apress en 2020. En 2021, j'ai décrit la bibliothèque à la communauté Ring pour la première fois. En 2022, la première présentation de Softanza l'appelait « le meilleur ami des programmeurs Ring existants ». La simplicité de Ring et son expressivité multiparadigme sont deux choses que Softanza garde, parce qu'elles n'ont jamais été le problème.

CE QUE DES PROJETS RÉELS ONT APPRIS

Trois choses, et ce sont les raisons du passage à Haro.

Les limites sont apparues. Des projets réels ont montré que Ring n'était pas adapté aux défis des applications modernes. Les limites sont en trois endroits : la performance, le typage fort et la facilité de construction. Les propres sessions du projet les ont mesurées, et le dossier garde les lignes avec leurs reproductions. Le

chapitre 27 imprime la mesure de la première : le moteur à moins de 2x d'un pair rapide du terrain sur chaque ligne, ce que l'ancien runtime n'a jamais été. Sa mémoire se tient à côté de celle de l'ancien runtime.

La direction ne pouvait pas être influencée. La direction d'un langage appartient à son propriétaire, et nous ne pouvions pas influencer celle du nôtre. Le projet a construit Ring++ pour résoudre ces trois difficultés sans quitter Ring, et il n'a pas reçu l'accueil que nous espérions.

La gouvernance est une propriété de conception. Une dépendance dont on ne peut pas influencer la direction, et dont on ne peut pas encadrer le comportement, échoue à la diligence que demandent les domaines régulés. La réponse de Softanza n'est pas l'indépendance pour elle-même, mais l'indépendance pour ses utilisateurs : un moteur écrit en Zig sans dépendance tierce, une machine virtuelle et un compilateur à nous, et des engagements que le projet fait déjà tourner comme des tests. Un noyau ouvert. Des chartes ratifiées en public. Chaque décision dans un registre daté. Chaque affirmation liée à sa preuve.

LE CHEMIN

Ring est le point de départ. Ring++ est le pont, en construction. Haro est la destination. Sous la bibliothèque se tient le moteur en Zig, quel que soit celui des trois qui est au-dessus : chaque fonctionnalité est implémentée une seule fois, dans le moteur, et le chapitre 27 dit la règle.

Aujourd'hui la bibliothèque tourne encore sur son premier runtime, version 1.27, et les blocs de ce livre tournent dessus. Haro emploie la syntaxe de ce premier runtime comme sa première facette, si bien que tout le code déjà écrit continue de fonctionner. Les gardes de la bibliothèque sont l'épreuve de cette promesse : Haro doit les faire tourner. Le chapitre 27 dit où en est le moteur, chiffres à l'appui.

REMERCIEMENTS

Softanza dit merci au langage dont il est issu, et va là où la souveraineté de ses utilisateurs l'exige.

Si ce livre est utile à votre organisation, je travaille avec les organisations sur ses sujets : exemplaires papier en quantité, ateliers et formations, conférences, conseil et réalisation de solutions. Écrivez à kalidianow@gmail.com.

Comment ces livres sont offerts : https://mayouni.github.io/ayouni/fr/offre/

Libre à lire et à partager sous CC BY-NC-ND 4.0 ; le code sous la licence de la bibliothèque.

https://mayouni.github.io/ayouni/fr/licence/

Declare, then govern.

Born in Africa. Useful to the World!

Vous utilisez un outil qui écrit du code. Vous lui demandez une fonction, ou une fonctionnalité entière, et il répond en quelques secondes. Vient alors la question qu'aucun outil ne résout pour vous : ce code doit-il entrer dans mon système ?

Le code est devenu bon marché. La confiance, non. **Le changement arrive plus vite que la confiance.**

CE QU'EST SOFTANZA

Softanza est une plateforme logicielle gouvernée. Vous déclarez ce que vous savez d'un monde, en texte clair ; un tribunal construit à partir de cette déclaration le garde ; et le modèle d'IA, quand il y en a un, vient en dernier. Elle est trois choses : une bibliothèque que vous appelez depuis du code, un moteur sous elle qui fait le calcul, et un langage, Haro, fait pour tenir les déclarations.

POUR QUI

L'artisan du logiciel : quiconque déclare ce qu'il sait d'un monde, et possède ce qu'il déclare. Le programmeur, l'analyste, le concepteur, le décideur, le formateur, et l'agent d'IA, puisque ce livre est aussi publié sous forme de données qu'un programme peut lire. Vous n'avez pas besoin de connaître Softanza. Il vous faut avoir écrit du code, et avoir ressenti le doute ci-dessus.

Chaque bloc de code de ce livre a été exécuté, et son verdict est imprimé sous lui.

MANSOUR AYOUNI

Il a écrit son premier programme à dix ans, au stylo sur du papier, à Regueb, un village du centre de la Tunisie qui n'avait pas l'électricité. La programmation l'a mené ensuite de la Tunisie au Niger et au Canada : des logiciels pour des cabinets d'avocats, des banques et la gestion des déchets nucléaires, des équipes de trois cultures. Il a conçu Softanza, dont il a été le premier artisan.

AYOUNI

PREMIER DES LIVRES AYOUNI

Libre à lire, sous toutes ses formes

mayouni.github.io/ayouni/books/softanza/fr/

ÉPREUVE DE TRAVAIL · CC BY-NC-ND 4.0