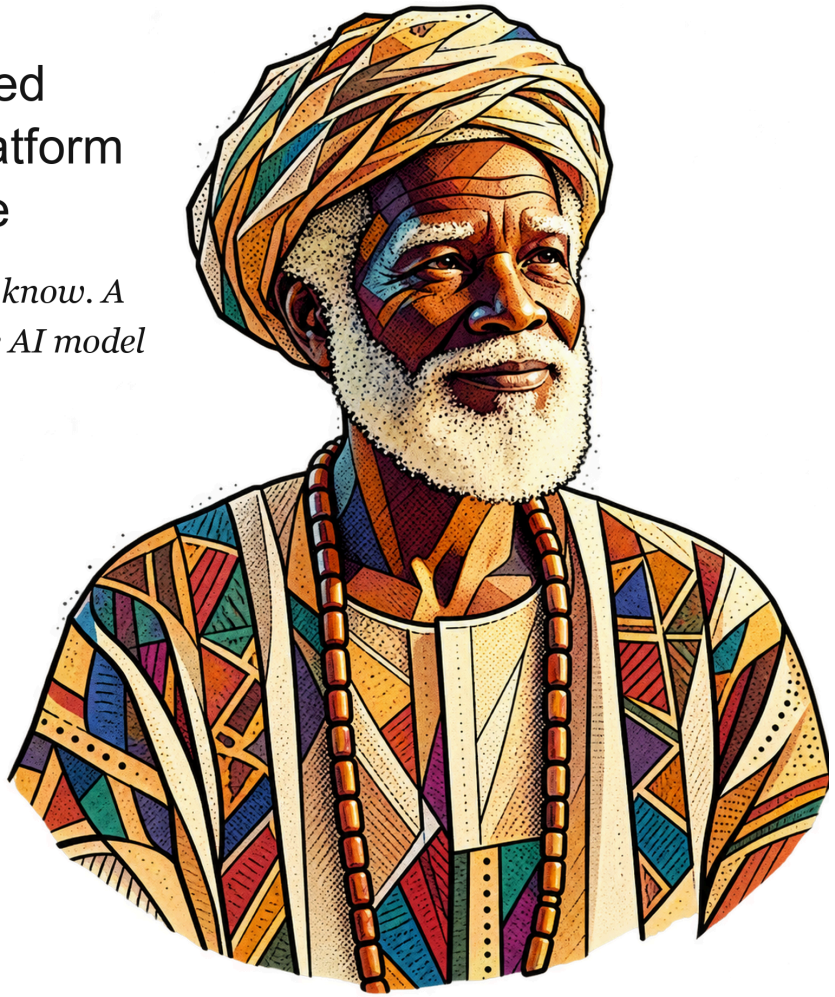
AYOUNI

Softanza

The Governed Software Platform of the AI Age

Declare what you know. A court keeps it. The AI model comes last.



MANSOUR AYOUNI

TUNIS · NIAMEY · PARIS · 2026

Softanza

Softanza

The Governed Software Platform of the AI Age

Declare what you know. A court keeps it. The AI model comes last.

Mansour Ayouni

AYOUNI

TUNIS · NIAMEY · PARIS · 2026

Proof copy: 42 pages in draft, 0 in review, 0 published, of 42.

Every block of code was run inside the library at commit 93d7a39bd: 285 of 285 blocks kept their promise. The verdicts printed are the run's, never rewritten.

Printed 2026-10-17. First of the AYOUNI books.

© Mansour Ayouni · AYOUNI

Contents

Preface SPINE 9

PART I · THE AGE, THE CRAFTER, THE METHOD

1 Change arrives faster than trust SPINE 17
2 The software crafter SPINE 27
3 The method, from first principles SPINE 43
4 The platform in one picture SPINE 56

PART II · PROGRAMMING, RETHOUGHT

5 Law: the code adheres to the way a person thinks, and the court judges it whoever wrote it SPINE 72
6 The mental model, and the metaphors 86
7 The writing style, and the seven goals 99
8 Functions as linguistic expressions, and scope 116
9 Patterns of thought: the Softanzuter, Reaxis and the engines 132
10 Programming becomes refinement 147

PART III · LANGUAGES, RETHOUGHT

11 Law: a language is declared and closed, and whatever speaks it, person or model, is judged by it SPINE 161
12 A language, declared 176
13 Natural, and executable: two directions, one discipline 195
14 Knowledge programming 215
15 Pattern languages 233
16 Seven languages, one door 250

PART IV · INTELLIGENCE, RETHOUGHT

17 Law: programmatic by default, the model last, admission over generation SPINE 269
18 The five registers, and conversations that ask 283
19 The domain language model 302
20 Agents that cannot hurt you 316
21 The court: promises, perception, security and performance 337
22 Compared, honestly 356

PART V · THE PLATFORM, ASSEMBLED

23 A world, a constellation, a platform, and the architect 372
24 Rehearse, then commit 394

25 Your users keep their medium 418
26 The environment 438
27 The engine beneath: three layers, one engine, Haro 459

PART VI · SOFTWARE CRAFTERS, FORGED

28 A restaurant in Lyon 479
29 A bank in Niamey 495
30 An organisation in the field 508
31 What the projects show, and what comes next 523

PART VII · WHAT COMES NEXT

32 Many minds on one body of work SPINE 543
33 The next line: a solution on its own engine SPINE 554

APPENDICES

A The register of innovations 567
B The fourteen house rules 574
C The twenty-two verbs and the constitution 579
D The glossary: words name states, never people 583
E The guards and fixtures cited, by chapter 590
F The narrations the book argues from 604
G The book as data, for the agent that reads it 612
H The project's history: from Ring to Haro 617

PREFACE · SPINE

Preface

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 1 of 1 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

A QUESTION YOU ALREADY HAVE

You use a tool that writes code. You ask it for a function, or for a whole feature, and it answers in seconds. Then comes the question that no tool answers for you: should this code enter my system?

You have always had ways to answer. You run the tests. The type checker agrees, or it does not. A colleague reads the change. Those ways were made when people wrote code slowly. Now one sentence can produce a thousand lines, and nobody has time to read a thousand lines for every sentence.

Code has become cheap. Trust has not become cheap. This book keeps coming back to one sentence, the one every chapter rests on: **change arrives faster than trust.**

THE IDEA, IN TWO MOVES

First you **declare**. In plain text, you say what a thing is: an application (I call it a *world*), a language, an AI agent, a whole product. Then you **govern**. From your declaration you get a judge, which I call a *court*. A court is a validator built from your declaration. It gives a verdict, *kept* or *refused*, and the reason, in the words of your declaration.

Every change goes to the court before it enters, whoever wrote it: you, a colleague, a script or an AI model. The model can still propose anything. It cannot decide anything. It is also asked last: whatever rules can decide, the rules decide first.

A DECLARATION, IN REAL CODE

Here is the smallest case. You add an AI agent to your system, a program that watches stock levels and reorders. You do not start with its code. You start with a short text file that says what it is: its name, how often it runs, what it may do, and whether its actions can be undone. That file is a declaration. You load it, and the court reads it before the agent can ever run. This agent's file leaves something out. What does the court say?

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.
```

✓ ran: printed what this block promises

Read it in three parts. The text between the quotes is the declaration. The next line loads it. Then `?` prints an answer, and the line after `#-->` is what the block printed when I ran it. The court answers `0`: refused. It names the rule and gives the reason: an agent must say what it covers, because the engine will not schedule an agent that does not, so the file is refused now and not at its first run. A few words of the

file: `pia` names the file format, `compensable` means its effects can be undone by a compensating action, and `law 18`, in the court's reason, is the engine's own number for its rule that an agent with no coverage is never scheduled.

You could write this check in your own CI. Here nobody had to. The court for agents ships with the library, so the rule is the library's, and the facts are yours, in the declaration. A one-off assert in a start-up script judges one fact. A declaration holds all the facts about the agent in one text, and the same court judges it whether a person, a script or an AI model wrote it. The rule can be yours too: chapter 2 has a rule you write in a few lines over two facts, that an actor is a language model and that it holds an effect, a power to act on the world such as sending a mail, and the court finds the violation in your own graph of agents. This example is small on purpose, and a schema can check it too. The book does not argue that a court checks more than a schema. It argues that one court, built from your declaration, stands in front of every actor and gives its reason. It does not read a thousand generated lines and call them good: it checks that a change fits what you declared, and chapter 24 shows a change tried out on a copy of your system before anything is committed. For a thing of your own, such as a language you invent, chapter 12 shows how you make the court, and says how far that is built. Chapter 1 tells a second scene, with a mail service, and chapter 2 runs this block again.

You will say that you already have tests, schemas and policy checks in your CI. You do. Today those rules live in many places: tests here, a schema there, a policy file elsewhere, each written by hand after the code. Here you state them once, in the declaration, and the court comes from it. Chapter 22 compares the two honestly, and where the field does better, it says so. A court judges what you declared. It cannot judge what you left out, and each chapter's limit says where that stops.

WHAT SOFTANZA IS

A governed software platform. You declare what you know about a world, in plain text; a court built from that declaration keeps it; and the AI model, where there is one, comes last. That is the cover's line, and every chapter runs it. In matter, Softanza is three things, and for each I say how far it exists.

- A **library** that you call from code. You write a few lines of Haro like the ones above, and you run them with the library loaded. The library holds the declarations and the courts made from them. Its source is at github.com/mayouni/stzlib, and every block of code in this book ran against a named commit of it. If your work is in Python or TypeScript, today you would use the library beside your code, not inside it: through one governed door it can run code in six other languages, Python, NodeJS (JavaScript), R, Julia, Prolog and C, which with the host make the seven of chapter 16.
- An **engine** beneath the library, written in Zig, a systems language. It does the actual computation, once and in one place, so that every language that talks to it gets the same behaviour. It runs today, and the library's domains are moving onto it one by one: chapter 4 says which still compute in the host language.
- A **language**, Haro, made to hold the declarations. For now it uses the syntax of its first facet, so all the code written so far runs unchanged. Its own executable is in development: it has a charter and a first experiment.

Chapter 4 draws the three in one picture.

WHERE IT STANDS

Softanza is not a research project, although it rests on deep research. Real projects shaped it: a restaurant, a bank and an organisation, and their feedback shaped its design. Parts of Softanza are in production. Other parts are in development, as is normal for a platform that is new and this ambitious. Each chapter says which is which.

WHAT THIS BOOK IS

This is a book of arguments, with running code. For each question about programming, about languages and about intelligence, I do four things. I give the answer the field usually gives. I give the answer Softanza takes. I give the decision I made, and what I refused. And I give the proof: the code ran, and the tests that

prove it are named. The third, the decision and what I refused, is the reason this book exists: the site shows what is, and only here will you find what was refused. I also say what the field already does well, so that you can compare.

It has seven parts and thirty-three chapters. You do not need to know Softanza or Zig. You need to have written code, and to have felt the doubt above. The project's website holds its evidence, with the proofs, and its course teaches the same ground by exercises. This book holds the argument.

Each chapter has six short doors, near its end. A door is a way in for one kind of reader: the programmer, the analyst, the designer, the decision-maker, the educator and the agent. The agent is an AI program, because the book is also published as data that a program can read. Enter by the door that is yours today.

HOW TO READ IT

Read nine chapters first. They carry the whole argument and can be read alone: the four of Part I, the last two, and three that each state a rule the book holds everywhere. Chapter 5: the code should read the way a person thinks. Chapter 11: a language is declared and closed. Chapter 17: a program is programmatic by default, which means rules and code decide first, and the AI model comes last. The reader has a button that shows only these nine. Come back to the others when one of them interests you.

Every chapter has the same nine parts, in the same order. The idea as the field holds it today. The first principle. The form it takes in Softanza, in code. The decision, and what I refused. The court: the named tests that prove the chapter's claim. The field, and what it does better. The limit. The six doors. An exercise, which you answer before you run it. Chapters 5, 11 and 17 open with one more short section, on what the AI age changes.

One more name you will meet. Softanza asks you the question you have to answer, in your own words, and refuses to guess the answer for you. I call that practice wise coding. Chapter 18 runs it, register by register.

WHAT TO TRUST

Every block of code in this book was run inside the library. Under each title the page says when, at which commit, and how many blocks kept their promise: how many printed what the line after `#-->` says they print. If a block prints something else, the build fails, and so does the page. The exercises are yours to run.

Two words come back on many pages. The *record* is the repositories, sessions and files this book ran against, at the commit printed under each title. The *study* is my own study of that record, made before the chapters were drafted. When a page says that the record holds no answer, it means those files, at that commit.

When a chapter says what is not finished, it uses three words and no others. *Built*: the code exists, a test exists, and a block ran. *Designed*: a document says so, and nothing runs. *Absent*: I looked and found nothing, and I say how far I looked. The project's website uses five names for the same things, and chapter 3 shows how they map to these three.

One more note. Haro's own executable is still in development, so the blocks marked `haro` run today on the runtime of the language the library was first written for, version 1.27, and each page says so. Appendix H, the project's history, tells how that came about. Inside the code, the library's own names, its classes, its files and its keys, still carry that language's name, and will until the library is retired from it; the prose does not.

ABOUT THIS DRAFT

The chapters were drafted with AI agents working from the project's own files, and every code block was run. Each page carries its stage at its top: Draft, written, its code run and judged, read by fresh readers; In review, while I review it; Published, once I approve it. Today every page is a draft. Where the files held no answer to a question, the page says so in one plain sentence and does not invent one. The French edition is a translation of the English drafts, and each French page says so. This preface was written after the chapters, and it stays unfinished until I write my

own. This is the first book of my imprint, AYOUNI. The second, *Refinement-Oriented Programming*, is about one discipline for software in the age of AI agents, told through one bank.

PART I

The age, the crafter, the method

1 · Change arrives faster than trust
2 · The software crafter
3 · The method, from first principles
4 · The platform in one picture

PART I · CHAPTER 1 · SPINE

Change arrives faster than trust

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 2 of 2 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Ask the industry what changed in programming these last years and you will get one answer, said with excitement or with dread: *machines write the code now*.

The picture is familiar. An AI model proposes a function, a file, a whole application. A person reviews it, or does not. Tools are added around the model to make it safer: linters that read what it wrote, tests it generates for itself, guardrails that catch the worst of it, a second model that judges the first. The model is at the centre. Everything else is a fence built around it after the fact.

I do not think that picture is wrong about the facts. It is wrong about where to stand.

2. THE FIRST PRINCIPLE

Here is the sentence I keep coming back to, and it is the oldest sentence in this book: **change arrives faster than trust.**

Code has become cheap. A thousand lines cost a sentence. What has not become cheap, what has in fact become the scarcest thing in software, is the trust that those lines do what their author, human or machine, says they do. Every team I have met in the last two years has more code than it can review and less confidence than it had before. The bottleneck of our age is not writing. It is admission: deciding what may enter a system and stay there.

So the question Softanza asks is not *how do we generate better?* It is *how do we admit?* Who judges a change, by what law, before it runs, and what is written down about the verdict so that someone can read it later and be answerable for it.

When you put admission at the centre, the picture turns around. The court is in the middle. The model is one proposer among others, and it comes last, after the person, after the data, after the formula, after the plain sentence. Nothing it says enters the world until the court has read it. That is not a fence built around a generator. It is a different building.

3. THE SOFTANZA FORM

Let me show you rather than tell you, by example, which is how I have always preferred to teach.

A solution declares that it depends on a mail service. While I program, I bind that dependency to a sandbox: an object that captures every message and sends nothing, so my tests can read what *would* have gone out. Free, offline, deterministic. Then I move the solution to production, with the sandbox still bound, the way people do at two in the morning before a demo.

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
```

✓ ran: printed what this block promises

The registry does not call it sound. In the emulated phase it answers one; in production it answers zero, and the next line says why: a fake must never ship. The fake is refused by name, and a service that was declared and never bound raises

the moment it is asked for, instead of silently doing nothing. I did not remember to check this. The phase decided what comes back, and the check is enforced, not remembered.

That is the whole philosophy in one small scene. Expression is free: bind what you like, while you like. Admission is governed: the gate reads the phase and the binding, and refuses.

Now a second scene, because the first one has no AI model in it and you are waiting for the model.

Take a restaurant as the illustration: an AI agent that knows its world proposes to place an order. In Softanza the agent does not place it. It proposes a plan. A workbench rehearses the plan on a twin of the system, a copy that holds no reference to the real one, and records every operation it would perform. A court reads the rehearsal against the rules the restaurant declared: who may order, what may be ordered, what a supplier must confirm. Only then does a governed actor commit, and the commit is recorded with its actor and its outcome. The guard that proves this sequence, `safeworld_narrated`, rehearses in a twin of the files, plans, and lets an actor commit; it holds seventy-six assertions as I write, none of them about a restaurant, and the site shows a recorded run of the same mechanism.

Notice what the model did and did not do. It proposed. It did not decide, did not write, did not send. It was a proposer beside the registers of answer. Three of them, an option chosen, a data structure given, a sentence spoken, are built today and open to the person before the model is called at all; two more, a formula written and an example shown, are not yet built (Chapter 18). I call this *programmatic by default, the model last*. It is the sentence under every chapter of Part IV.

And a third scene, the smallest, because it shows the same law at the level of a single line. The platform is judged on what its code looks like and what it answers:

HARO

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

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

✓ ran: printed what this block promises

The question is spoken as it is thought: find all, remove duplicates. And the two lines under the two questions are not my commentary. They are promises the library kept on the night this chapter was built. If it stops keeping them, this page fails. The same law, admission before trust, applied to a paragraph.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

Every chapter of this book carries this section, and I promised in the preface that it is the reason the book exists. So let me be concrete about the decisions behind this first chapter, because they were not comfortable.

I refused to put the AI model at the centre. When the AI models arrived, that was the obvious move, and I had the pieces to make it: a natural layer, a knowledge graph, an agent class. I could have wired a model into each and shipped a Softanza that "talks". I did not, and I wrote the refusal into the architecture as a law: the mind is programmatic by default, and a model may only propose. I gave up some fashion by that choice. I think I kept the platform.

I retired the language I was born on. Softanza was written for years on a small, warm scripting language that I love and that I pleaded with, long before the LLM era, to make executable natural language its first concern. It did not, and that is its right. In 2026 I ruled that the language would not be carried forward: the runtime beneath Softanza is being replaced by one we own, written in Zig, a charter and a spike so far, wearing the old surface so that every line I wrote can keep running, under a new name, **Haro**. The old surface stays because the code is the asset; the old runtime goes because sovereignty over the runtime is one of the commitments I made. That decision was not free. It is in Chapter 27 with its numbers, including the one I did not want: our runtime is slower than the one we measured against, by a factor between one and two, on every row. I recorded the loss in the estate memo of 13 September. A book that hides that is not worth your trust, and neither is a platform.

I refused to let anything into BASE without a court, and I failed. The three layers of the library, CORE, BASE and MAX, were meant to keep the foundation small and the innovations above it. Under the pressure of time I let everything go into BASE: ninety-three percent of the code sits there today, and the dependency rule is held at the engine and broken at the inheritance. I say this on

the first chapter, not the last, because the discipline I am about to preach for thirty-two chapters is one I have broken, and the study of the record caught me. That is what a court is for, and the court for the layers is still only proposed.

I refused a second embedding. When someone asks "what about Python, what about JavaScript, what about my language", the answer is one door, to be governed, and never another host runtime sewn into the platform. The record's seven languages, the host and six runtimes behind one door, were chosen for seven layers of a solution, and Chapter 16 counts them. The door is the answer to "what about language X", and it is one answer, not twenty.

What was refused is as much a part of Softanza as what was built. You will not find the refusals on the site, because the site shows what is. You will find them here, in every chapter, under this heading.

5. THE COURT

What proves this chapter? Three things, and I give them to you by name so that you can check them rather than believe me.

The service registry's refusal of a fake in production is a guard, `service_registry_narrated`, in the system plane. The AI agent's propose, rehearse, judge and commit is `safeworld_narrated`, seventy-six assertions, in the agentic plane, beside `governance_crossing_narrated`, in the system plane, for the crossing itself. The code blocks on this page were run by the book's own runner inside the library on the night this page was built, and its output is shown as it came.

Three guards for a chapter of philosophy may seem few. They are the right three, because they are the three places where philosophy meets a verdict.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

I will not pretend the field has nothing. It has, in the last three years, built the most capable code generators in history, and every day I use them. The industry's answer to the trust problem is real work: generated tests, review bots, policy engines, sandboxes for AI agents, structured outputs, a second model to grade the first. Where the field leads, and it leads clearly, is in the generators themselves. Softanza does not build the large generators and does not compete with them; where it trains, it trains small models bound to one domain, whose every answer will be checked by a grammar. The field also leads in scale: its review tools read volumes of generated code that nothing in Softanza has been run against.

Where I believe Softanza stands alone is in the order of things. The field adds admission to a generator. Softanza adds a generator to a court. In the field, an AI model writes in your language and tools check it afterwards; here, the model may only write in a language you declared, whose grammar is closed and whose court is meant to be born with it, so that, for the languages whose grammar is compiled today, an invalid sentence cannot be spoken at all. In the field, the dialogue with a

system is a prompt; here, the system asks, and the answer can be an example, a formula or a sentence before it is ever a model. In the field, the record of a decision is a log; here, it is a transcript with a speaker and a certainty, answerable forever.

Those are claims, and each has its chapter and its guard. I list them here so that you know where the argument is going, and so that you can hold me to it.

7. THE LIMIT

BUILT	the court that admits or refuses a change, for a person and for a machine
ABSENT	a customer platform running on the Softanza engine

The honest limit of this chapter is also the honest limit of the whole platform, and I put it on the first page on purpose.

Softanza has been used. A bank in Niamey runs an assessment platform whose building shaped Softanza, under licence, attested by the installation guide we delivered. A restaurant in Lyon runs a platform for neighbourhood commerce whose first demo its owner made alone, on his own phone, with a messaging app as his support line. An organisation in Niger has its hub designed. Their lessons are in the code, by file name, in Part VI.

But the customer platforms run today on ordinary stacks. The bank's is a Python application; the restaurant's is a set of single-file web worlds; the organisation's hub is not built. Their declarations are being written as Softanza's own, so that moving them onto the Softanza engine is a change of engine and not of plan. That move is the next step. The court is built, and the platforms exist; the next milestone is the first of them running on the engine. That is the next line this book will carry, and Chapter 33 is about nothing else.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You will keep your editor and your language. What changes is that every claim you make about your code can carry a promise under it, and a court will read it.
The analyst	Your rules become data a person can change, and every change is judged before it lands and recorded after.
The designer	The interface is judged too: twenty-two verbs, one hundred and twenty-two rules, and a verifier that refuses what a machine can check.
The decision-maker	The question you ask every vendor, "who decides, and what is written down", has a mechanical answer here, and it is the same answer for a person and for a machine.
The educator	The tutor asks and never answers first. That is a law of the platform, not a feature of a product.
The agent	You are welcome here, and you will be judged. Appendix G is yours.

9. EXERCISE

Take one function an AI model wrote for you this week. Under it, write the line it must print for one input you care about, in the form you have just met:

HARO

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

✎ not run: an exercise, written for you to run

Run it. If the line matches, you have your first promise. If it does not, you have your first finding, and you have learned more about that function in a minute than the model's explanation told you. Do not fix it yet. Chapter 2 is about who you are when you do.

WHERE THE EVIDENCE LIVES

ON THE SITE	forged
ON THE SITE	agentic
RULINGS OF THE STUDY	12.10, 12.11, 12.15, 12.28, 12.29

PART I · CHAPTER 2 · SPINE

The software crafter

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the word "crafter" from the field's craftsmanship and from its two answers that define the person by a tool, low-code and the coding agent. I say what the site means by it. I show the six doors, each with a line or two that run, and the three cases this book is for: a restaurateur in Lyon, Softibank, a bank in Niamey, an organisation in the field. I say what I kept, the programmer first; what I added, the AI agent, with one power withheld; and what I refused, "maker" and "builder". The principle is one sentence that joins the site's and the glossary's: a crafter is whoever declares what they know about a world, and owns what they declare.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field already has a word for programmers who care about their work: craftsmanship. A book of 2001 gave the idea its name, and a manifesto of 2009 gave it its slogan. Since then "software craftsman" has meant a professional who holds their code to a standard higher than the one the job requires. That is general knowledge, not a reading of the record. It is a good movement. It is also a movement of programmers, about programmers' standards.

The field's two other answers to "who makes software, if not programmers" define the person by a tool. The first is low-code: business people building applications on a vendor's visual platform, the "citizen developer" of the analysts. The second is the coding agent: an AI model that reads files, edits them and runs commands,

steered in plain words, with "vibe coding" as the name for keeping whatever it writes. Both answers share a shape. The person is defined by a tool, and the work lives where the tool lives. When the vendor changes its prices or its AI model, the person's work changes with it. That too is general knowledge, not a reading of the record.

The next section says what the site means by the word.

2. THE FIRST PRINCIPLE

The site gives the definition in two sentences. Since 8 October 2026 they say "crafter"; before that they said "maker". The glossary says that a crafter is whoever is declaring, at the moment they declare. The site says that a crafter owns what they make. I keep both, in one sentence: **a crafter is whoever declares what they know about a world, and owns what they declare: the artefact is theirs, in plain text, and it does not expire.** Skill does not enter the definition. The word "craft" suggests it, and I keep the word: the person is defined by the act and by ownership, not by programming skill. I hold this as a view, not a result.

3. THE SOFTANZA FORM

I was the first crafter of Softanza, and not for a reason I would have chosen. My wife runs a music club where people learn to sing classical Arabic songs. Every Sunday morning she sends me the list of songs to prepare, and I search for their lyrics. The lyrics found online repeat every phrase as the singer performed it, and she reminds me, every time, to remove the duplicates. I wrote `RemoveDuplicates()` to stop doing it by hand. Is its family still in the library, and what does it say of a list with three teas in it?

HARO

```
o1 = new stzList([ "tea", "rice", "tea", "fish", "rice", "tea" ])
? o1.ContainsDuplicates()
#--> 1
? o1.NumberOfOccurrence("tea")
#--> 3
```

✓ ran: printed what this block promises

The family answers: the list holds duplicates (1), and `tea` occurs 3 times.

I write about this in an article on what the library was built on: I introduced its features when a real problem asked for them, or when I anticipated they would meet programmers' needs. This one came from a problem: the crafter comes first, and the feature after.

Since 2022, I have presented Softanza not by its anatomy, its modules and classes, but by who you are and what you are trying to do. The 2022 deck had five audiences, each with a recipe of three to five steps. On 9 October 2026 the site had seven doors, one per kind of crafter. The seventh, for a team that ships an application, was added that day, and this chapter keeps the six it was drafted with. Each door says what you declare, what you get, and where to start. Here is what each of the six declares, in a line or two that run.

The programmer declares intent in a library that reads like a sentence. You have just seen it.

The analyst declares the data of a world, and the engine computes the view. Here, three regions of Niger with the population the 2012 census counted:

HARO

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

✓ ran: printed what this block promises

The engine drew the view from the data alone: `Maradi │ 3402094` is one of its rows.

The designer declares a picture as a program, and gets the same picture every time:

HARO

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

✓ ran: printed what this block promises

The picture is the program's output: a boxed `Softanza` with round corners.

The decision-maker declares what each actor in the world may do, a person or an AI agent, and the court enforces it. An agent is a small text file. What does the court say to one that does not say what it covers? Load it:

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.
```

✓ ran: printed what this block promises

The court refused it! Its reason is printed: `[pia-coverage @ coverage] an agent must say WHAT IT COVERS`. Here "law 18" is the engine's rule that an agent with no coverage statement is never scheduled, and `pia` names the format of the file, as its first line shows.

The educator declares in the learner's own language. A sentence in Hausa runs: it asks for a text holding `'hello'`, turned to capitals. Does the engine count the letters of an Arabic word rather than its bytes, and does it know its script?

HARO

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

✓ ran: printed what this block promises

The Hausa sentence came back as `HELLO`. The Arabic word counts `4`, its letters and not its bytes, and its script is `arabic`.

The agent is the sixth crafter, the one the platform's 2020 presentation could not draw, in the words of the vision I ratified. An AI agent declares itself in the same kind of file, and it is welcome, with one thing withheld. An agent whose mind is a language model may propose; it may not hold an effect. What happens to a file that gives it one?

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).
```

✓ ran: printed what this block promises

The llm actor `rogue` does not get its effect! The court answers `REFUSED`, and says when: `enforced at CONSTRUCTION, not merely audited`. A "pi-gate" is the gate where a governed actor commits.

Now the three cases this book is for. They are not personas. They are people and organisations with documents, and Part VI tells each of them in full. Here, a page each.

A restaurateur in Lyon. He owns a couscous restaurant, and the project is a platform for neighbourhood commerce. The project's notes say it plainly: he is not a programmer. The project began with an offer in May 2024 and then two years of rich specification that produced nothing he could use. What came out of that failure is the rule I have worked by since. Deliver small, fast and useful, and let the foundations mature in parallel without ever blocking a delivery.

On 4 July 2026 he answered a scoping report in his own words and reversed the order of the work: customer first, restaurateur next, supplier last. The project's architecture was redrawn as a constellation of worlds, using the library's design, and the first amendment closed on 4 August 2026.

A constellation is a graph of worlds. A world is one application; a bond is a declared relation between two worlds, for one named act, and nothing crosses without one. Chapter 23 shows all three run; here it is only the shape the project took.

The project counts eleven iterations and a campaign of twenty-two simulated tests behind its interface rules. In his own report the speed of ordering is his first, non-negotiable priority. And on 19 August, after a day of tests in the field with two phones and a messaging app, the notes call him the project's co-architect.

The first demonstration, on 15 August 2026, was his alone: the record's checklist plans it for the owner by himself, with his own phone and a messaging app as the support line. The measurements he sent over a messaging app are those of 19 August.

One of his needs is carried by a law of the library: try payments before paying anyone for an account. What does the registry say to a fake payments service while you rehearse, and what does it say when you go live? Run it. The registry is named once, and each line inside the braces speaks to it: chapter 5 calls this form the scene.

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
}
```

✓ ran: printed what this block promises

It answers `1` in the emulated phase and `0` in production, and says why: `still bound to a SANDBOX in a production phase -- a fake must never ship`. A fake in production is refused!

A bank in Niamey. A commercial bank under the rules of the central bank of the West African monetary union is the licensee of an organisation-assessment product I built beside the library. It is installed on its own internal network, under that licence. The installation guide delivered to the bank is the document that attests it. The lesson it left in the library is that an organisation can be judged by its own chart, under a regulator's rules. Declare a chart with a chief executive and an internal audit that reports to him. What do the regulator's rules say of it?

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" ] ] ]
}
```

✓ ran: printed what this block promises

The chart fails, with `"issuecount", 3`: three findings from a chart of two positions! Each is a rule with a number an auditor can read: `BCEAO-001: No Board of Directors found` is the first. The record shows the bank as the licensee of a product I built; it does not say who at the bank declared anything.

An organisation in the field. An organisation working in Niger, with a head office in Niamey and bases in the field, asked for a hub. Its own specification preferred a low-code platform built on a large vendor's office suite. My offer of 5 August 2026 was chosen instead. The design study lists four gains, with ownership first among them. The organisation owns its vocabulary and its rules, changes them without programming, pays no licence per seat, and can hand everything to a

programmer of its own country. The design study was built with its teams between 22 September and 8 October 2026. Two of its principles are the book's in small: the default is to work offline, and the AI agent proposes while a person decides. That rule is data the organisation attaches to its own domain model. Does that model judge itself, with an LLM actor in it that holds an effect?

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
```

✓ ran: printed what this block promises

`RulesAreSound()` answers `0`: the domain model finds its own `writer`, an LLM actor holding `effectful`, in violation of the rule.

The three cases are a restaurateur who is not a programmer, a bank under a regulator and an organisation that chose ownership over a vendor. The restaurateur and the organisation are crafters by the record; the bank is the licensee of a product, and the record does not say who there declared anything. The restaurateur and the organisation did not wait for the software industry, and each is offered ownership of what is built for them.

4. THE DECISION, AND WHAT I REFUSED

I kept the programmer first. The vision of the platform, which I ratified, says it in one line: the first audience stays the programmer in a small team; the agentic era did not replace that person, it multiplied them. I held that AI agents do not make programmers obsolete.

I added the AI agent as a crafter, and withheld one power. The agent is the sixth reader of every Softanza language: both a builder and a user. It gets a workbench and a court. It does not get the capability to commit.

I refused "maker", and I refused "builder". "Maker" names the workshop movement, the soldering iron and the 3D printer, as general knowledge has it. The site's headline used it until 8 October 2026. "Builder" names someone who assembles what a plan says. Neither says what I think programming is. It is handwork, done with care, on something that is yours: it is craft. I call the person who does it a crafter.

I refused to count a customer without a document. The site lists a project as a reference only when a document proves it, and quotes no number of countries and no number of programmers. A customer's name goes on the site on my word; its standing as a reference needs the document. A lesson is listed only when a file of the library carries it.

I refused to present the library by its anatomy. Since 2022, the front door is who you are and what you want to do, and the anatomy is placed under each step. On 9 October 2026 the site kept seven doors, and the journeys behind them existed as pages, nine on that day, each step carrying its stage word. That every step runs in order under a guard is designed; building it is the next step.

The guards that prove these claims, and the one still in development, are in the next section.

5. THE COURT

`agentfile_narrated` proves that an agent is a file judged when it is loaded: one refusal per gate, the missing coverage among them, and the agent whose mind is a language model, holding an effect refused by the same rule the graph uses. `service_registry_narrated` proves that a sandbox in production is a violation. `graphrule_object_narrated` proves the rule as an object attached to a model. `semantic_resolver_narrated` holds the Hausa path as its last scenario. At this commit it is in development: its run stops after 29 assertions, before the Hausa ones, on an error raised from a string-splitting function, and finishing that run is the next step. So the page counts its assertions and does not claim its verdict; the Hausa block ran on its own. `00_orgchart_narrated` proves the chart's positions, reporting lines and vacancy tracking; the BCEAO finding printed above comes from a sibling file, `14_bceao_governance.ring`.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

Coding agents lead in hands. They already read, search, edit and run. What they lack, by the site's own account, is a way to know an unfamiliar interface without guessing, to judge their work against rules and promises, and to present a change as an intention a person can review. Softanza's judges are built; the door between them and a coding agent is designed, and building it is the next step.

Low-code leads in installed base and in everything around it, as general knowledge has it. The organisation in the field keeps its vendor's suite for its accounts, its documents and its notifications. The crafter's platform sits beside the field's tools, not in place of them.

The messaging apps lead where the crafters already are, and the record of the restaurant does not yet make one his medium. The restaurateur's tool before the project was a spreadsheet; his own report of 4 July 2026 says so. The line that names messaging apps is the label of an option in the scoping form, and not his answer. No page of the site names those apps, and no conversational medium is a target of the platform yet. Chapter 25 is about that direction.

And the educator's door is at an earlier stage than the others. The course's editions in French, Arabic and Hausa are drafts that a reviewer has yet to read, and the first institution to adopt it is still to come.

The next section says what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	the agent file and its court; the registry that refuses a fake in production; rules as data on a model; a regulator's rules judging an organisation chart; the natural layer in Hausa and Arabic
DESIGNED	a guard that fails when a step of a journey page breaks, each step run in order; the door through which a coding agent asks the library to judge its work
ABSENT	the platforms of the three customers running on the Softanza engine; a crafter's own medium, the messaging app on their phone, as a target

What is built: the agent file and its court; the registry that refuses a fake in production; rules as data on a model; a regulator's rules judging an organisation chart; the natural layer in Hausa and Arabic.

What is designed: a guard that fails when a step of a journey page breaks, each step run in order; the door through which a coding agent asks the library to judge its work.

What is absent: the platforms of the three customers running on the Softanza engine. Today the restaurant's platform is a set of web worlds, each a single file; the bank's is a Python application; the organisation's hub is a design. Their

declarations are being written as Softanza's own. And the crafter's own medium, the messaging app on their phone, is not yet something a world can be declared to.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You declare intent in a library that reads like a sentence, and an engine computes under every call. You stay the first audience.
The analyst	You declare your organisation's entities, rules and flows in plain text, and the rules judge the model.
The designer	You declare a picture, a map or a screen as a program, and the first rule of an interface can come from a real trade: for a restaurateur, the speed of ordering.
The decision-maker	You declare your world, your governance, and what each actor may commit, and you own all of it, in plain text.
The educator	You declare a course in your learners' language, and every exercise is checked by running, in the site's words. The editions in French, Arabic and Hausa are drafts a reviewer has yet to read.
The agent	You declare yourself in a file: what you cover, how reversible your acts are. You get a workbench and a court. You do not get the capability to commit.

9. EXERCISE

Declare one AI agent for a world you know: a shop, a class, an office. Write its file: what it covers, how reversible its acts are, when it runs, one skill. Before you run it, write the promise you expect under the question below. Then run it.

HARO

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

✎ not run: an exercise, written for you to run

If the court refuses, read the rule it cites and change your file, not the court. Then make your agent a language model and give it an effect, and read why it is refused. You will not find the corrected file in this book.

I left this one in your hands. Take the graph-rule block of section 3, the one for the organisation in the field, and change one line: the writer's capabilities become `[ "inference" ]`, without `"effectful"`. Run it and read the last answer.

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()
```

✎ not run: an exercise, written for you to run

Then compare it with the answer printed in section 3, and say what the rule read to give each. Section 3 shows its two conditions. When you can say it, you will know which line of the domain model the rule reads.

Chapter 3 is the method, from first principles, and the sentence its method returns to is this: you declare a world in a language made for it, and you govern how that world changes.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	offering
ON THE SITE	customers
ON THE SITE	forged
ON THE SITE	coding-agents
RULINGS OF THE STUDY	12.2, 12.12, 12.13, 12.25, 12.29, 12.30

PART I · CHAPTER 3 · SPINE

The method, from first principles

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 5 of 5 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I show the method every chapter after this one follows. I take "from first principles", the field's slogan, and give it a narrow meaning: a concept, the field's form, a redesign in running code, and a court that judges it. I show the smallest rethink I know, the empty string, and then the two acts, declare and govern, in a few lines each. I say what I refused. First, a feature without a need, and general-purpose code as the meeting ground between people and machines. Then a title that looked back, the grid alone, and a reading called a census. The whole method returns to one sentence: you declare a world in a language made for it, and you govern how that world changes.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In our industry, "from first principles" is a slogan. It means to reason from the bottom, not by analogy. You will hear it in talks and read it in launch posts, and you will rarely be shown what it produced.

The field does have a method for the decisions behind software, and it is a good one. A team writes a short record for each decision it takes, made up of the context, the decision and the consequences. Many teams call these architecture decision records, from a blog post Michael Nygard wrote in 2011. The field also has a method for proving what a function does: an example written under the function, which the tools can run as a test. Python has offered this since 2001 with its doctest module, which runs when someone invokes it, and Rust runs the

examples in its documentation as tests by default. And when the field compares one tool with another, it draws a grid: rows of features, one column per tool, ticks and crosses.

The field has three forms; the method returns to one sentence.

2. THE FIRST PRINCIPLE

Here is the sentence the whole method returns to: **you declare a world in a language made for it, and you govern how that world changes.**

Two acts. Everything else in this book is one of them, or the two together.

3. THE SOFTANZA FORM

"From first principles" can be checked. In this book it has a narrow meaning. Take a concept everyone uses, write down how the field holds it, redesign it, and leave code that carries the redesign and a court that judges it. The site's history page is titled *From first principles*, and the library has been on GitHub since 12 March 2022. The phrase is not decoration here. It is a work order.

Let me show you the smallest rethink I know. Ask Python how many times the empty string occurs in the word "text", and it answers five: one before each letter and one at the end. Ask it whether "text" contains the empty string, and it says yes; JavaScript says the same. I asked Python 3.13 and Node 22, on 8 October 2026. That is a convention of implementers, not a fact anyone would say aloud. Softanza states a rule instead, and the rule has a name in the narration that argues it: emptiness is uncountable, and emptiness is uncontainable. What does Softanza answer to the same two questions? Run it:

HARO

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

✓ ran: printed what this block promises

It answers `0` and `FALSE`. Two lines, and a decision you can disagree with. That is what a rethink looks like when it is honest: small enough to read, explicit enough to refuse.

A reading of the record, made in October 2026, counted forty-six such concepts, in seven families. The first two are the paradigms, such as scope-oriented programming, knowledge programming and reactive programming redesigned, and the paradigm engines, such as truth, quantifiers and time. The other five are patterns and reaction; expression and perception; data, time and reasoning; system, delivery and operations; and the library that knows itself. Forty-six is a reading and not a census; the site's own register page, read on 9 October 2026, says forty-seven. Each concept on it has an article or a design document of its own, and that was the test for entering the list.

Now the two acts, each in a few lines.

Declare. A kind of value is declared in a sentence, and every verdict against it is computed and explains itself. Why can't -4 be a 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')
```

✓ ran: printed what this block promises

Notice the last line: `no: -4 is not positive (constraint 2 of 'score')`. The verdict names the rule it applied, by its number! A verdict on a declared kind that cannot say why is not admitted in this house.

Govern. A declared constraint, once enforced, refuses a change before it lands. Here the string "hello" must stay lowercase. Does the update to "HELLO" land?

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
```

✓ ran: printed what this block promises

The update was refused! The object stands as it was: it still answers `hello`. Declaring a constraint and enforcing it are two acts, and the guard behind this block checks that a constraint you only declared stays passive.

Governing also has a side that faces the person. The system asks, and it does not let you close a world that is not complete. What does it say when you try to conclude anyway?

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.
```

✓ ran: printed what this block promises

The goal was declared: every dish must say what it contains. The gap between the goal and what is known generates the question, and the question carries its reason. Concluding while two gaps remain is refused! The printed line is `Can't conclude: 2 gap(s) remain -- the wise-coding loop is not done. Ask the next question.`, and its last words tell you what to do next. Chapter 18 is about this conversation. Here it is enough to see that governing is not only refusing; it is asking.

One more rethink, because the paradigm engines are part of the method too. Truth, in the language beneath Softanza, is a matter of empty and non-empty, and a list of words counts as true. Softanza keeps that truth by default and lets you extend it by declaration. Once "wrong" is declared an item that makes a list false, is a list that holds it still true?

HARO

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

✓ ran: printed what this block promises

The host's answer is unchanged: `TRUE`. The extended answer reads your declaration and says `FALSE`! "Truth becomes a matter of policy", the narration says, and the policy is written where you can read it.

So this is the method, in the form you will meet in every chapter after this one. A concept as the field holds it. The one principle the record returns to. The redesign, in running code, with its promise under each block. The decision, and what was refused. The court that proves it, by name. The field, and where it leads. The limit, in three stage words. The first, **built**, means the code exists, a guard exists, and a block ran; it is the site's own *built*. The second, **designed**, covers what the site

calls *in construction*, *specification* and *ratified proposal*. The third, **absent**, covers what the site calls *named*, a name with nothing behind it yet, and what nobody has named. Five words of the site, three of the book. Then six doors, one for each kind of reader, and one exercise.

And the book is judged the way the library is. Every block in it ran inside the library on the night the page was built, at the commit printed at the top of the page. A page whose promises are not kept does not build.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

There is no single record of a decision called "the method". The method was decided in pieces, and the pieces are in the record.

I refused to add a feature without a need. I wrote it in an article of the library, which carries no date. "I introduce new features only when they solve a tangible problem or when I anticipate they'll meet developers' needs." First principles without that rule produce a museum. With it, a rethink has to earn its place: someone had a problem the field's form did not answer.

I refused general-purpose code as the meeting ground between people and machines. The vision of the platform says it in one line, which I ratified. Humans and AIs meet in declared languages, small, closed and judged, and not in general-purpose code. Declaring is the first act because it makes the second possible. A court can only judge a sentence of a language it knows.

I refused a title that looked back. When this book was first proposed, its title was *Programming by Heart*, the slogan of the 2020 presentation. I turned it down as outdated: it named a stance, and the book had to name what was rethought. The next proposal was *Declare, Then Govern*. The cover carries instead the sentence the preface gives: declare what you know, a court keeps it, the AI model comes last. The two verbs moved to the back cover, where you will find them. They organise every chapter; they did not need to be its name.

I refused the grid alone. The library's documents hold dozens of comparison grids, and in them Softanza's column wins almost every row. A reader who sees only those grids is right to ask for more than a grid. The honesty in the record is elsewhere: in the sections of the field studies titled "what the field has that Softanza lacks, honestly", and in measured numbers. When the runtime beneath Softanza was measured again in September 2026 against the one it was compared

with, the earlier headline did not reproduce. The loss was recorded the same day, in an internal memo. That is the rule this book follows: a loss is printed beside the win, in the same section.

I refused to call a reading a census. The forty-six concepts are a reading of the record by one session, not a list I ruled row by row. Appendix A prints the register of forty-six rows as the study gives it, with the book's own column, as a working appendix: the rows are not ruled one by one.

Those are the five refusals; now the court that judges the blocks above.

5. THE COURT

The three blocks of the two acts stand on two guards. `modality_narrated` proves that a declared kind can be inspected, that a verdict names the constraint that failed, and that an unknown kind never silently allows. It also proves that an enforced constraint refuses a bad update before it lands, while a declared one stays passive. `wisecoding_narrated` proves the conversation. It proves gaps computed from the declared goal, a question that says why it is asked, answers admitted, and a law refusing through the conversation and reaching a person. And it proves that concluding with open gaps is refused. Its sister, `wisecoding_rich_narrated`, carries the richer cases of the same loop.

The rethink of emptiness stands on `612_dealing_in_emptiness_in_softanza.ring`, a library test written as scenarios, each line stating the result it expects. The rethink of truth stands on its narration and on this book's run, and a guard for it is the next step. The library's file for it, `04_truth_in_softanza.ring`, sets the list of false words by assignment and never calls the function, so it does not prove the block.

The court has been named; next, the field, and where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in adoption. Decision records are a light standard that many teams use, with templates and tools around them. The decision element of this book exists in this book and in one study of the record, and not yet as data in the library. Examples in documentation can be run by the standard tool of Python, and Rust runs them by default. The promises of Softanza are judged by the house's own runners, which no outside tool reads yet. And the field compares by grid, as the record's own pages also do; a grid alone is a start, which is why section 4 asks for the loss beside the win.

Where I think Softanza stands apart is in one place only: the rethink is carried all the way down. I hold this as a view, not a result. The decision is written, the redesign runs, the promise is judged at build, and the stage word says how far each piece has gone. A decision record does not run. An example does not say what was refused. Here they are one object.

7. THE LIMIT

BUILT

	the court behind every block of this chapter but the truth one, and the stage words the site prints on every product
DESIGNED	the register of rethought concepts as data, with a guard that fails when a concept has no proof
ABSENT	a page that shows each rethought concept with a run proof; a test file for the paradigm engine of section merging

What is built: the court behind every block of this chapter but the truth one, and the five stage words the site prints on each product.

What is designed: the register of rethought concepts as data, one row per concept with its article, its code, its guard and its stage. Beside it, a guard that fails when a new concept arrives without a row. Both are a specification today.

What is absent: a page that shows each concept with a run proof; the site's register page, read on 9 October 2026, lists each with its evidence. A test file for the engine module of one paradigm engine, section merging: it has none of its own, though its merge methods run in `base/test/string/289_content.ring` and `max/test/stzListOfPairsTest.ring`. And a few rethinks where the article goes a step beyond the code of today. The article on quantifiers asks whether all your marks `AreGreaterThan` the average, and that method is spelled differently in the library today. Its friendlier quantifiers, some, most and few, draw at random, so they carry no promise today. The register is a reading, and a reading can miss a concept that lives in only one article.

That is where this chapter stops; six doors follow, one for each kind of reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Every claim you make can carry its promise. An enforced constraint refuses a bad update at the object's single update point, the one `UpdateWith()` reaches: `Update()` for a string or a number, `_SetContent()` for a list.
The analyst	Your rule is a declaration, and its verdict names the rule that failed, by number.
The designer	The stage word is part of the interface. A product that does not exist yet says so, in a word you can check.
The decision-maker	Ask every vendor for the loss beside the win. Chapter 1 prints one, with its date: the runtime beneath Softanza trails the one it was compared with by a factor between one and two on every row, measured on 13 September 2026; chapter 27 gives the rows.
The educator	Each rethink has an article that argues it: the concept, the field's form, the redesign. That is a curriculum, ordered by family.
The agent	You learn this platform from the platform, and you are corrected by it. A verdict that says why is written for you as much as for a person.

9. EXERCISE

Choose one default of your own language that you never questioned: what the empty string counts to, what counts as true, what "most" means. Before you run anything, write in one sentence the rule you think is right. Then write two promises in the form you have met: the line your language prints today, and the line your rule must print.

HARO

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

✎ not run: an exercise, written for you to run

Run both. I will not tell you which rule is right. If your rule and your language disagree, you have found a decision that someone took for you, and you can now take it yourself.

I left this one in your hands, on purpose. Take the block of section 3 that governs a string, change one word, `EnforceConstraint` for `AddConstraint`, and run it as it stands below. I print no output for it.

HARO

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

✎ not run: an exercise, written for you to run

Look at what it prints: whether the update was refused, and what the object holds. Then read section 3, just after the block, and section 5, on the guard that checks it: the page says why. When you have both lines, you will know why a declared constraint is not an enforced one.

Chapter 4 draws the platform in one picture, under one idea at every altitude: nothing is believed, everything is judged.

WHERE THE EVIDENCE LIVES

NARRATION	`stzstring-emptiness-narration.md`
NARRATION	`istrue-softanza-truth.md`
NARRATION	`stz-built-on-real-world-need.md`
ON THE SITE	history
ON THE SITE	principles
ON THE SITE	estate
ON THE SITE	compare
RULINGS OF THE STUDY	12.19, 12.22, 12.29, 12.30

PART I · CHAPTER 4 · SPINE

The platform in one picture

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 8 of 8 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I draw the whole platform in one picture, read from the bottom up: the engine, the layers, the areas, the worlds, the crafters, and beside every floor its court. I walk it floor by floor, with one block at each floor that you can run. Then I say what I refused along the way: computation in the faces, a flag to choose a layer, mini-applications. The picture is summed up in one idea at every altitude: nothing is believed, everything is judged.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field draws a platform as a stack of boxes. The kernel at the bottom, a runtime and its libraries in the middle, applications on top, and sometimes a column at the side for security or tools. Inside one codebase the same picture becomes a rule about dependencies: an inner layer knows nothing of the outer ones, and the arrows only point inward. Rust's three standard crates, one for bare metal, one for memory, one for a full operating system, are the cleanest version I know. Three depths of the same language, chosen by what the target offers.

For applications, the field's picture is a host. A large application embeds a runtime and runs small applications inside it, which share its services and its store. In both pictures the judge is outside: a test suite beside the code, a review beside the store. In the pictures I know, neither draws the court as a column standing at every floor.

Softanza draws that column, and one idea holds the whole picture.

2. THE FIRST PRINCIPLE

The stack of the platform is summed up, in the vision I ratified, in one idea at every altitude: **nothing is believed, everything is judged.**

3. THE SOFTANZA FORM

Here is the whole platform in one picture, read from the bottom up. Each floor stands on the one beneath it, and each floor is judged by a court beside it. The numbers in it are not typed by me. The build reads them, when this page is made, from the library and from the site's map of its areas.

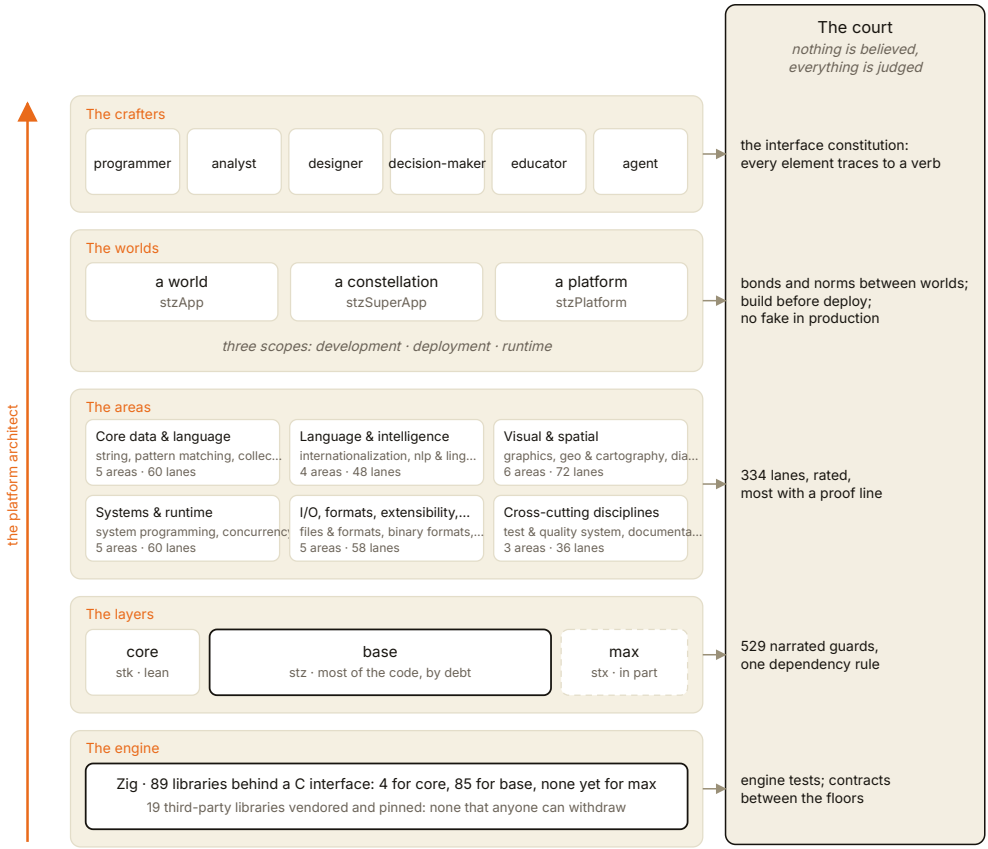


Fig. *The platform in one picture. Each floor stands on the one beneath it; each is judged by the court beside it; the architect works across all of them. The numbers are read at build from the library's engine, its test tree and the site's Atlas.*

Let me walk it, floor by floor, with one block at each floor that you can run.

The engine. At the bottom is an engine written in Zig: a set of libraries behind a C interface, one per domain, grouped by the layer that loads them. By the engine's design rule, the engine does the computation; the library above it gives it a

vocabulary. A small example makes the division visible. Ask the language beneath Softanza for the length of the Arabic word for peace, then ask the engine. What does each one count? Run it:

HARO

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

✓ ran: printed what this block promises

The language counts bytes and answers `8`. The engine counts letters and answers `4`, and it knows the script: `arabic`. Four letters, not eight!

The engine also has its own door, beneath every class. Numbers too large for the machine's arithmetic are added exactly, by the engine itself:

HARO

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

✓ ran: printed what this block promises

The engine answers `100000000000000000000000000000000`, the exact sum.

The layers. Above the engine, the library is cut in three layers, by depth of capability. Core, with the prefix `stk`, is lean and talks to the engine directly, for small devices and small binaries. Base, `stz`, is the rich layer, where the vocabulary and its forms live, enough for most solutions. Max, `stx`, is meant for the strongest innovations. A layer is chosen by the file a program loads. Does the same verb mean the same act at core and base? Run it:

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
```

✓ ran: printed what this block promises

Both answer `1`: the same verb, the same act, at two depths.

The areas. Across the layers, the library is mapped by the site's Atlas into areas grouped in bands, and each area into lanes. Each lane is rated from emerging to strong, most of them with the line that proves it. The Atlas says what exists, and how well. Another principle of the engine's design is that a verb means the same act on every structure that admits it. Does it hold for one pair, a string and a list? Run it:

HARO

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

✓ ran: printed what this block promises

On the string, `FindAll` answers character positions: `[ 1, 10 ]`. On the list, it answers item positions: `[ 1, 3 ]`. The same act, on two structures.

The worlds. Above the library, the unit is not a module but a world: an application is a living world of meaning, with its things, its relations and its rules. A world is declared. What does it say of itself when you ask it to explain? Run it:

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)
```

✓ ran: printed what this block promises

It answers `WORLD FieldVisits  lives in: memory (not persisted)`, then lists its being and its relations. `Live()` answers `[FieldVisits] is live -- 2 thing(s), 0 flow(s), 0 reaction(s), 0 goal(s); 0 proposal(s)`. Two things, and no flow, no reaction, no goal. The world says what was declared, and no more.

Worlds compose into a constellation. Here a restaurant and its produce supplier are two worlds, and a bond between them declares the one crossing that is allowed. The three `Gov` lines declare the crossing's risk, grant it to the restaurant and set the restaurant's authority; the guard behind this block shows that without them even the declared crossing is refused. What becomes of a crossing that no bond declares? Run it:

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'.
}
```

✓ ran: printed what this block promises

The declared crossing answers `1`. The other answers `0`, refused! The constellation gives its reason in one line: `no bond declares 'raid-kitchen' from 'resto' to 'supplier'.` Every other crossing is refused with its reason.

The commons is the shared ground of a constellation, what every world of it stands on. It holds an identity, a session and a store. It is a world like the others, world zero, and not a node of a special kind. Chapter 23 shows it and says how far the code has come in that direction.

And a platform, `stzPlatform` in the figure, is the envelope around a solution: its parts, the systems they run on, and the order in which it is built, rehearsed and deployed. You declare the solution it carries as a `stzPlatformProfile`, and you will see it refuse to deploy before it is built in chapter 33. Here is what its delivery planner, `stzDelivery`, does with the layers, for the web: a part of a solution ships only the engine groups it needs, and only those that are Softanza's own. The phone below asks for four capabilities. How many engine groups does it ship? Run it:

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" ]
```

✓ ran: printed what this block promises

Two, not four: `aggregation` and `solver`! The phone's own system handles one of the other two, Unicode, and Softanza ships the other, Regex, as a construct in the target's own language.

The crafters. At the top are the six doors of chapter 2: the programmer, the analyst, the designer, the decision-maker, the educator and the agent. And one person works across every floor, from the engine to the organisation: the platform architect. The record names that role in three of its documents, and no other role crosses every band of the Atlas.

The court. Beside every floor stands its judge. The engine has its tests and the contracts between the floors. The layers have the narrated guards. Most lanes in the areas carry a proof line. The worlds have their bonds and norms, the order of build and deploy, and the refusal of a fake in production. The crafters' interfaces have a constitution of verbs and rules that every element must trace to. And above all of it, the platform's threat model lists its guarantees, each tied to the guard that proves it. Its own guard fails when a row names a guard that no longer exists.

Parts II to IV of this book are rethinks, and each of them is a box of this picture. Part V assembles the picture in use.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused computation in the faces. The engine is the product. The language above it adds beauty, not computation: thousands of methods that read like sentences, and the algorithms beneath them in one place. The rule is that every feature is implemented once, in the engine. The migration is in development. The layer contract lists number, list, object, graph, stats and natural as still in the host language the library was first written for.

I removed the old framework, with no fallback. The library sat on a large C++ framework for its text, its dates and its files. When the engine replaced it, every path through the framework was removed: no fallback, no feature flag kept in case.

I chose sovereignty as a verdict on each dependency, not as "every line ours". The engine vendors and pins a set of third-party libraries, the figure counts them. The record defines the word in one line: nothing anyone else can withdraw. That is the test. A library whose source is pinned in the tree passes it.

I refused a flag to choose a layer. The first way of choosing a layer was a flag. It could never work, because the language beneath loads every file it sees at compile time, so all three layers loaded on every call. The failure of one was the failure of all. The flag was removed rather than kept as a switch that silently does nothing. The reason was written where the next reader would stand: a layer is chosen by the file you load.

I let everything go into base, and I say so. Under the pressure of time, the domains landed in base. The engine, the planes such as security and performance, and the customers moved forward meanwhile. That was the agile choice, and the next step is to place each domain in its layer, in plain sight. So the figure draws base widest, and chapter 27 gives the numbers.

I refused mini-applications. The field's super-app is a host that runs small applications. Softanza's is a governed graph of worlds: worlds, not mini-apps; bonds and norms, not a container.

Five guards stand beside the floors; the next section names what each proves.

5. THE COURT

`00_stzapp_narrated` proves a world declared and made visible. `06_stzsuperapp_narrated` proves the constellation: worlds joining, the commons as world zero, crossings gated by norms with the bond named as the reason, a retired world breaking its bonds. `full_scope_model_narrated`, the architect's common ground, proves the solution modelled first, the scopes that refuse what a target forbids, and the order of build and deploy. `01_platform_narrated` proves the platform's duties. `threat_model_narrated` reads the platform's threat model and fails when a guarantee names a guard that no longer exists.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

Rust's three depths are enforced by its compiler and used every day on bare metal. Softanza's rule is held at the engine and at the entry files, and not yet carried to where base should build on core, as section 7 shows.

The field's text libraries lead on locale: Softanza's own map names the international components' locale layer as the clear gap of its strings, and names the bet behind it: its locale area leaves those components out on purpose, with no collation, no plural rules and no time zones.

At the scale of real users, the field's super-app hosts lead today. The record names them as the model it refused and measures nothing against them, so I will not either.

And the Atlas itself prints what is still to come for Softanza. The binary area is a plan for now, and its shipped code is the next step; the plugin, foreign-function and embed plane of extensibility is mostly planned; the system area has few lanes rated strong so far. I hold this as a view, not a result. A map that prints its stages is the only kind worth reading.

7. THE LIMIT

BUILT	the engine and its libraries; the base layer; worlds, constellations and platforms with their guards; the engine subset a part ships, for the web
DESIGNED	the max layer loaded by its entry file; the base layer building on the core one; a loader for one domain of base alone; a court that checks the layers against their contract
ABSENT	a whole program and its engine packed into one file; a customer platform on the engine

Now I put the three stage words on the picture, built, designed and absent.

What is built: the engine and its libraries, the base layer, worlds, constellations and platforms with their guards, and, for the web, the engine subset a part ships.

What is designed, or built and not reachable. The max layer's own classes are written and loaded by no entry file. What does the runtime answer when you ask for one? Run it:

HARO

```
try
        o3 = new stxString("one and one")
catch
        ? cCatchError
done
#--> Error (R11) : Error in class name, class not found: stxstring
```

✓ ran: printed what this block promises

The runtime answers: `Error in class name, class not found: stxstring`. The class is written, and no entry file loads it!

The layer contract says the base layer builds on core, by inheritance or delegation. It does neither for the common domains: the two share the engine and implement those domains twice. The core layer itself is in development against the engine today. Its string and character classes call engine names that no engine file registers, as my study of the record reports. The block in section 3 shows lists only. A loader that brings one domain of base without the others is not built. A court that reads the layer contract against the tree and fails on drift is a specification, proposed in that study. What is built is two implementations sharing one engine; the contract's line that base builds on core is designed.

What is absent: a whole program and its engine packed into one file with no dependency, and a customer platform running on the engine.

8. SIX DOORS

Each of the six doors reads this picture from where it stands.

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You choose your depth by the file you load, and one verb means the same act on a string and on a list.
The analyst	The Atlas tells you, lane by lane, where the platform is strong and where it is emerging. Build on the strong lanes.
The designer	Your interface has a court of its own, the constitution's verbs and rules, in the same column as the code's guards.
The decision-maker	Ask two questions of every floor: what stands beneath it, and who judges it. The engine's answer to the first is a written list of pinned libraries.
The educator	The picture is the book's map. Each rethink in Parts II to IV is one box of it.
The agent	The picture exists as data you can read, the Atlas as a file, and every floor you write to has a court that will read you.

9. EXERCISE

Draw your own solution the way this chapter draws Softanza: floors, and a court beside each. For each box, write the file that declares it and the guard that judges it, and put a stage word on it: built, designed or absent.

Then declare two parts of it with `stzPlatformProfile`, give the profile to a `stzPlatform`, and before you run anything, write the line you expect when you ask it to deploy before you build it. Run it. A box with no court beside it is your first finding.

I left this one in your hands. Section 7 says the core layer's string classes are still in development against the engine, and the block of section 3 shows you lists only. Take the block of section 7 and change one word: `stxString` becomes `stkString`.

HARO

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

✎ not run: an exercise, written for you to run

Run it, and read what the runtime answers. Then go back to section 7 and find which name is called, and who should have registered it. When you have solved it, you will know why the layer contract's line that base builds on core is called designed, and not built.

The next chapter leaves the picture for the code written inside it, and states the first law: the code adheres to the way a person thinks, and the court judges it whoever wrote it. Its principle is one sentence: what you think is what you write.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	architecture
ON THE SITE	platform
ON THE SITE	platforms
ON THE SITE	areas
RULINGS OF THE STUDY	12.2, 12.10, 12.12, 12.13, 12.15, 12.23, 12.29

PART II

Programming, rethought

> *The code adheres to the way a person thinks, and the court judges it whoever wrote it.*

5 · Law: the code adheres to the way a person thinks, and the court judges it whoever wrote it
6 · The mental model, and the metaphors
7 · The writing style, and the seven goals
8 · Functions as linguistic expressions, and scope
9 · Patterns of thought: the Softanzuter, Reaxis and the engines
10 · Programming becomes refinement

PART II · CHAPTER 5 · SPINE

Law: the code adheres to the way a person thinks, and the court judges it whoever wrote it

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the two things the field calls coding style and ask the question it does not ask: whether a line follows the order in which its writer reasoned. I run the founding example in the nearest form that still runs, then the mental model on a list, the verb's forms, the named parameter, the scene and the history of a chain. A promise stands under every question. I say what I let go, the library's own implementation, and what I refused to let go, the style of the code written with it. I show the court that exists today, over the library's own classes, and the one that is only designed. The motto has not changed since 2020: what you think is what you write.

WHAT THE AI AGE CHANGES HERE

For years, most of the code written with Softanza was written by one hand, and the hand was mine. The style of the library was the way I wrote. Then the machines arrived, and today most code here is written by AI models, as the design of the intelligence plane says it is everywhere, in one sentence: most programming today is done by agents and language models, not humans.

That changes what a style is. A style that lives in one person's habits dies when that person stops writing every line. It has to become something a machine can be held to, and something a machine can check, or it thins away. In this house it did thin, and I will show you where. So this movement of the book, Programming rethought, opens on a law that has two halves. The second half exists because of the age: the code adheres to the way a person thinks, and the court judges it whoever wrote it. By the court I mean the machine that judges code against the house's rules; today it judges only the library's own classes, and section 5 says how.

There is a dependency to say aloud. This book follows the order of its cover, from the familiar to the new: programming, then languages, then intelligence. The record's own logic runs the other way. The first act is to declare a language; programming happens inside a declared language; intelligence is governed by both. The vocabulary's forms are already declared in a spike of Haro, as its charter's entry of 3 October 2026 records. The usage that this chapter describes, the scene, the chain, the named parameter, the promise under each line, is meant to be declared too. So read this law knowing that Part III holds it up.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In our industry, "coding style" means two things, and neither is about how a person thinks. The first is layout and naming, enforced by machines: Python's style guide since 2001, Go's formatter since 2009, and the formatters that every modern project runs before a commit. They settle arguments about spaces and capital letters, and they are good at it. The second is the fluent interface: methods chained so that a line reads almost like a sentence.

The age of generated code added a third practice: check after. A model writes, the tests and the linters read what it wrote, and the model tries again if they complain. The line itself is judged for its shape and its result. Whether it follows the order in which the writer reasoned is a question I know no tool to ask.

2. THE FIRST PRINCIPLE

My motto for this has not changed since 2020: **what you think is what you write.**

Section 3 shows the motto at work.

3. THE SOFTANZA FORM

The founding slide put the whole idea on one page. On the left, a person's four steps of thought: I take the word "softanza"; I turn it to uppercase; I get its characters; I join them with a space. On the right, the same four steps as one line of code, as if the text on the left had been pasted and run.

That exact line no longer runs. Its third link, a method that turned the characters into a list of strings in one step, is not in the library today. It survives only in archived files and in one scratch sample, as `CharsQRT`. I tell you this first because a law chapter that hides its own founding example would not deserve your trust. What is the nearest form that runs, with the four steps still visible, and what does the library offer as one dedicated verb? Run both:

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
```

✓ ran: printed what this block promises

Both forms print `S O F T A N Z A`. The idea is intact; the fluent link was lost. A request goes to the library to restore it.

The mental model is the same idea at the level of a problem. You say the problem in plain words. You select the object that holds the data. Then you ask, in order: does it contain what I care about, how many, where? Then you act on those positions, or you call the verb that does it for you. Take a list with empty strings in it. Does it contain them, how many, and where? Run it:

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" ]
```

✓ ran: printed what this block promises

The list answers `TRUE`, then `3`, then `[ 2, 4, 5 ]`, and after the act it reads `[ "A", "~", "B", "~", "~", "C" ]`. Contain, count, find, act. Each line is one step of thought, and the names are the words you would use to say it.

The verb's form says what happens to the object. What does each form do to it? Run them on one string:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

`Removed` printed `HARO`, and the object still read `HAxxRxO`! `Remove` then acted on it, and `Content()` read `HARO`. The chain with Q ended on `H A R O`. By the library's convention, the past participle gives you a changed copy and leaves the object

alone. The plain verb acts on the object. The verb with Q acts and hands the object back, so the next step can follow.

Do you need to remember which argument comes second? A parameter can be named, and the name reads as English:

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" ]
```

✓ ran: printed what this block promises

The text prints `I ♥ Haro` and the list `[ "fish", "rice", "tea" ]`. The names `:With`, `:AtPositions` and `:And` say which argument is which.

How do several questions about one thing read as a list? They are asked inside its scene, a block of braces around the object. The object is named once:

HARO

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

✓ ran: printed what this block promises

The scene answers `1`, then `3`, then `[ 1, 10, 20 ]`. The same word counted `1`, then `3` once `:CS = FALSE` was said! The object was named once, and the three questions read as a list.

Can the steps of thought be read back? An object lifted with history keeps the result of each step of its chain:

HARO

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

✓ ran: printed what this block promises

The run printed `[ 12500, 13000, 11500, 23, 46 ]`. Five values, the starting one included, and the chain ends at 46! The article that first told this promised four values, without the first, and a final value one unit short, 45. The guard behind the block states today's rule, on a string chain: the initial value, then each step's result. The book takes the run's answer, not the article's.

The same steps can be written as a scene, the form I hold as the basis of the style: the object is named once, and each line inside the braces speaks to it.

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 ]
}
```

✓ ran: printed what this block promises

The chain says it in one sentence, and the scene says it as steps. The book uses both: the sentence where the thought is one, the scene where it is a procedure. The classic form names the object on every line, and many readers think in it:

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()
```

✓ ran: printed what this block promises

Three forms, one answer. A fourth is the near-natural sentence, `Naturally("...")`, which chapter 13 runs on text; on a number it answers `Unsupported object type` at this commit, and that is its next step.

One word on `@@()`. On the former host language, `?` prints a list one item per line; `@@()` asks for the list's expressive form, `[ 12500, 13000, 11500, 23, 46 ]`, the one a person writes. I have asked that Haro print that form by default; then `@@()` leaves the examples.

Notice the line under every question. It is not a comment. It is a promise, what the line must print. In this house it has been the form of an example from the first day: a short story, a few questions, and the answers written under them. That is the first half of the law, made visible: the code follows the order of thought, and each step says what it expects.

Now the second half. A rule a person must remember is folklore. The library turns its house rules into validators, so that a person or an AI agent can check a class before it lands. The rules are fourteen, and section 5 counts them; five of them read the graph of the code, the structure the library builds from a class's text. Here a class breaks the house's convention that a verb ending in Q returns the object. This rule needs the return statements, which the graph does not model, so it runs as a text pass. What does the court say of the class? Run it:

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
```

✓ ran: printed what this block promises

The court answers `FALSE`, and names the rule: `q-returns-object`. It did not ask who wrote the class! That is the whole of the second half. And the promise under each question is read by a machine too: the library's runner judges a promise as a claim about what the line printed. The house's oldest form of example is also its newest form of test.

The chapters of this part prove the law, each from one side. Chapter 6 tells the mental model and the four metaphors that replace loops: walk, check, yield, perform. Chapter 7 tells the writing style as fourteen rules of its own, not the fourteen house rules of the court you have just seen, and the seven design goals as its architectural half. Chapter 8 tells the forms a function's name can take, the deep form among them, and scope named in the verb. Chapter 9 tells patterns of thought: the machine that reacts to what it recognises, reactive programming in plain words, and the timeline and the grid. Chapter 10 tells how programming becomes refinement.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I let the library's implementation go. The code the models wrote inside the library is accepted as good. It does not look like mine. Its local variables carry underscores on both sides, hundreds of times per thousand lines. They are a safeguard. The language beneath Softanza lets a library's local variable overwrite a caller's global, and the convention prevents it. I did not ask for the old style there, and I would refuse a campaign to impose it.

I refused to let the style of the code written with Softanza go. Every line a programmer, a person or an AI agent, writes with the library must take the historic style. That style is the scene, the chain one step per line, the named parameter, the verb's form, and the promise under each question. That is the line I drew when I saw what had happened.

What had happened is measurable. My study of the record counted the promise under each line, per thousand lines. In the short recipes the house keeps, there are nearly a hundred. In my articles, seventy-five. In the classic tests, fifty-two. In the guards the sessions wrote, under one. They assert in another form, a call that names the expectation. They adopted it because the bare comparisons of the older suites printed 0 or 1, and that does not tell a block that failed from a block that never ran. That gap is closed now; the runner reads the promise as a claim. The study that counted them recommends one form for new test blocks: the classic form, a title, the questions and the promises under them. The converted blocks are left as they are. The record holds that recommendation, and no ruling of mine on it.

I refused to rewrite code when the language changed its name. When the language beneath became Haro, the code in my articles was renamed and nothing else. The forms are the asset; the name is a label.

The forms of the vocabulary were made data, and three old features were refused as grammar. In Haro, the forms a verb can take are declared once and generated. In the language's spike, the library's shipped names are judged against that declaration. Its charter's entry of 3 October 2026 reports 584 of the 595 shipped spellings of the string class in agreement. The design of that work refuses three things the record counted as features: names that tolerate a misspelling, parameters in any order, and method names in other human languages. The founding deck counted the last two; the narration of the forms counted the first. Misspelling tolerance and free order stay in the library, in part; the multilingual names survive only in an archived file. None of the three is part of the grammar a machine is held to. The refusal is written in the language's principles file, as its seventh principle, marked designed. The book prints it as the design's, and the record does not say it is mine.

I chose goals before laws. The 2022 design held seven goals, expressiveness, flexibility, reliability, consistency, human metaphors, practical abstractions and manageability, and every feature served one. The operating rules of today are laws paid for by incidents, which say how and not why. Chapter 7 puts the goals back beside the laws.

5. THE COURT

`coderule_narrated` proves that a house rule is a rule over the code graph and that the findings come out on planted code. It proves that clean code passes, and that the library's house rules are fourteen. `testoor_promises_narrated` proves that the promise is judged by a machine in the house's own form. Each rule is planted both ways: the lie it must catch, and the truth it must not accuse. `05_containsemptystrings` is the mental model's steps on the same list, a classic block kept as a guard.
`466_keeping_the_history_of_updates_of_a_softanza_objec` proves the history of a chain.

Section 6 sets the court beside the field.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in enforcement. Most mainstream languages have a formatter and a linter, and most projects run them before a commit or in continuous integration; that is general knowledge, not a study of the record. The court that would judge the style of a program written with Softanza is a specification; the one that exists judges the library's own classes.

The field leads in reach and in people. There is a labour market for its tools and none yet for this one, and the field's generators are the largest that exist.

The field also leads, today, over Softanza's own pages. The study that measured the record found that the site's code blocks carry no named parameter and no scene: a reduced dialect of the library, shown to every visitor. An AI agent that learned from the site would learn that dialect. I did not measure the site's blocks again. A style that is not shown is not learned.

Where I think Softanza stands apart is in what the style is tied to. A formatter judges the shape of a line; a fluent interface makes a line pleasant. Here the line is tied to a step of thought, and the verb's form is a contract about the object. The promise under the line is the example, the test and the documentation at once.

7. THE LIMIT

BUILT	the forms of the vocabulary in thousands of method names; the house rules as a court over the library's own code; the promise read by a machine in the house's own form
DESIGNED	a court of style for the code a programmer or an AI agent writes; the writing grammar declared in Haro, so an AI model can only write in it
ABSENT	any machine that judges today the style of a program written with the library

What is built: the forms of the vocabulary, in thousands of method names; the house rules as a court over the library's own code; the promise judged by a machine.

What is designed: a court of style for the code a programmer or an AI agent writes with the library. It would read a positional argument where a named one exists, a question with no promise, a chain on one line past two steps. And the writing grammar declared in Haro, so that an AI model's output is held to it before it is written.

What is absent: any machine that judges, today, the style of a program written with the library. The founding slide's chain does not run as written. In this book the discipline carries the name of its motto: what you think is what you write.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your lines follow your reasoning: select, contain, count, find, act. Each question carries its promise, and a court reads it whether you wrote it or an AI agent did.
The analyst	The method starts from your problem in plain words. The keywords you use become the names of the steps.
The designer	The verb's form is an interface contract: by the library's convention, the past participle does not touch the object. A court for that rule is designed, not built.
The decision-maker	I hold this as a view, not a result: a style a machine can be held to can be reviewed goal by goal, not line by line. It is the only reading one person can give to hundreds of thousands of lines.
The educator	Teach the discipline whole, and in order: formulate, then express, then be judged.
The agent	You will be corrected by the library, not by folklore. Write the question, then the promise under it.

9. EXERCISE

Take the list `[ "tea", "", "rice", "", "fish", "" ]`. Write the questions of the mental model in order, one line each: does it contain empty strings, how many, where. Then one act, with a named parameter. Under every question, write the line it must print, before you run anything.

Run it. Then ask an AI agent to solve the same problem, and judge its answer with two questions only: does each line follow a step of thought, and does each question carry its promise.

I left this one in your hands, on purpose. The Basket class of section 3 fails the court because `AddQ` returns `42`. Change that one value to `This` and run the question again:

HARO

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

```
? StzCodeIsClean(cCode)
```

✎ not run: an exercise, written for you to run

Do not stop at the court's answer. Ask whether the class, as it stands, follows the way a person thinks, and what the court read to answer you. I do not give you the verdict: section 6 says which court exists, and section 7 says what it does not read. When you have settled it, you will know why a court that passes a class has not yet judged the style of a program written with the library.

The next chapter is the mental model, and the metaphors. Where you would write a loop, it asks you to name the move you are making instead: walk, check, yield or perform.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	learn
ON THE SITE	craft
ON THE SITE	principles
RULINGS OF THE STUDY	12.2, 12.3, 12.14, 12.16, 12.17, 12.21

PART II · CHAPTER 6

The mental model, and the metaphors

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 7 of 7 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the seven steps of the mental model, of which chapter 5 ran the middle four on a list, and run them on a string, in the same order. Then I come to the loop, the place where the steps repeat, and to the four metaphors the library gives you instead: the walker, the checker, the yielder and the performer. I show the condition that all four carry inside an argument, the W condition the engine reads as data, and its WF twin, which runs host code. I say why I made the metaphors a design goal, why I replaced `eval` with a language the engine owns, and where the code of the articles this chapter rests on is not yet runnable at this commit. The principle under all of it is the chapter's own: when you would write a loop, name the move you are making, walk, check, yield or perform.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field has one tool for repeating a thing, the loop, and a family of cousins that hide it: the iterator, the comprehension, the stream with its map, filter and reduce, the lambda that carries a condition. They are good tools, and they all share one habit. They are written in the language of the machine's motion: an index that starts at zero, a counter that is incremented, a variable that accumulates. A person who reads the loop has to recover the question it asks.

The field also has a vocabulary of metaphors for programming, but it keeps them in the documentation. The code stays a loop.

2. THE FIRST PRINCIPLE

Say the problem in plain words, ask the same questions in the same order of any object, and when you would write a loop, name the move you are making: walk, check, yield or perform. The site's Craft page says it, as I read it on 9 October 2026: "A loop hides which move it is making. Softanza names the four moves, and a task is decomposed into them instead." That sentence is the site's, not mine; the sentence above is this chapter's principle.

3. THE SOFTANZA FORM

The mental model is seven steps, and they are the order in which a person asks. Define the problem in natural terms and take its keywords. Select the object that holds the data. Ask whether it contains what you care about. Ask how many. Ask where. Act on those positions, or call the verb that does it. And the seventh: celebrate, and repeat. In chapter 5 you ran those steps on a list with empty strings. They do not belong to the list. Do they hold on a string? Here they are, in the same order:

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
}
```

✓ ran: printed what this block promises

Contain prints `TRUE`, count prints `2`, find prints `9`, and act leaves `You and Me`, with the same suffix that says whether case matters. The three questions you ask of a string are the three you asked of a list.

When the steps repeat, the loop appears, and this is where the library gives you four metaphors instead. Each one names a move.

The Walker moves along a route. A walker is a map of its whole journey, built when it is created. It holds the positions it will visit, the positions it will skip, where it stands and what remains. What does it know before it takes a step? Ask it:

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 ]
}
```

✓ ran: printed what this block promises

`Walk()` is one step, and it prints the path of the move, from where it stood to where it landed: `[ 1, 3 ]`. A loop does not hold its route as a value you can ask. A walker does, before it moves. What does it do at the edge of the road? Beyond the end, when the walk is underway, it stops in silence. Before the start, it refuses:

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.
```

✓ ran: printed what this block promises

Ask for ten steps from a walk that has four and you get the four: `[ 1, 4, 7, 10 ]`. Walk again and it prints `[ 10 ]`, the position it stands on, with no error. Ask it to walk backward from the start and it refuses! Its own words: `Can't walk backward! Already at the beginning of walkable positions.` The article argues that this mirrors how we walk in the physical world. You stop when you hit a wall behind you, and that is an error. When the road ahead ends, you stop in silence.

The Checker asks yes or no. All of them, any of them, or where, with the condition written as a string, the W condition, W for where:

HARO

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

✓ ran: printed what this block promises

One item fails the test, so `CheckW` prints `FALSE`; a number is in the list, so `ContainsW` prints `TRUE`; and `FindW` says where, `[ 2 ]`.

The Yielder produces new data from old, and leaves the old alone. The Performer is its opposite number: it changes the items where the condition holds. Which of the two touches the list? Put them side by side on the same list and look:

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" ]
}
```

✓ ran: printed what this block promises

Look at the two `Content()` outputs, before and after the `PerformW` call. The yield returned `[ "HARO", "PYTHON", "RUBY" ]`. The list was untouched! The perform changed the list: `[ 1, "HARO", 3, "PYTHON", 5, "RUBY" ]`. The verb tells you which, so a design can say "this step only reads" and mean it.

Now the thing that all four share: a condition travels inside an argument. It is a small string, kept in a variable if you want. It goes in the same place for find, for items and for contains. And it has a twin that takes a real function:

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 ]
}
```

✓ ran: printed what this block promises

The two agree: `FindW` with the string and `FindWF` with the function both print `[ 2, 4, 6 ]`. `ItemsW` gives the items themselves, `[ 8, 12, 9 ]`.

The string form is the W condition; the function form is WF. They look alike, and the difference matters. The engine reads a W condition as data, in a small language of its own: literals, `@item`, arithmetic, comparisons, and a short list of functions. It cannot call your code. So what happens to a condition that tries to? It is not run:

HARO

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

✓ ran: printed what this block promises

Nothing was printed! The positions came back empty, `[ ]`, and `W()` shows the condition for what it is, data: `[ "where", "{ @item > 5 }" ]`. A W condition can be stored, shown to a person, and approved before it runs, because it cannot escape. The WF twin runs host code, and the library's own table says it is not safe with input you do not trust.

The code of the article "Bridging Minds and Code" does not run at this commit, as section 7 says.

The three models of that article, the semantic, the mental and the expression, are one idea in three places. The semantic model is the world of the program as knowledge: an apple is a fruit. The mental model is what you just did: create an object, store data in it, find things inside it, act on the positions. The expression model is the code that says it: select the object, type the name of the action, continue the flow of actions, convert the result into another object. Keep doing so until you have the final answer. So I tell them in words.

The next section gives the decision, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I made the metaphors a design goal. The 2022 presentation lists "human-centric metaphors" as one of seven design goals, and its four features are these four. A goal is what makes a feature justifiable or refusable. I hold this as a view, not a result. So I did not leave the metaphors as a nicety of the documentation. They are one of the reasons the library has the shape it has.

I replaced `eval` with a language the engine owns. The condition seat used to be compiled by evaluating the string as code, with a more expressive but slower extended form on the side. I removed both. The engine now reads the condition in its own small language, and the library keeps the host-code form as a separate verb, WF, with its warning. I refused arbitrary code from text, and I refused to split speed from expressiveness: the extended form is retired because the plain one does everything it did.

I kept the seven steps as written. When the course was designed, a reading of the technical core found five steps: select, contains, count, find, apply. The course teaches the seven as the source narration has them. Nobody renumbers the source.

I let the edge rule be a design principle. Before the start is an error, and beyond the end is ignored. The narration gives its reason in a sentence: it mirrors how we walk. The record does not date it as a ruling, so I tell it as the library's design.

The court comes next, and its guards do not cover every line above.

5. THE COURT

The four metaphors have no single guard. The yielder has a narrated one, `00_yielder_narrated`: four map operations, three filter operations and a sum. The walker has `597_walker_basics`, named walkers and what they walked, and `495_walking_until_and_until_before`, a walk that stops on a condition, including the item that stops it, or not; both test the list's named walkers and walks, through `StzListQ`, not the `Wk` walker of the blocks above. The checker has `611_checkw`, which says a check is false when one item fails. The performer has `621_perform`, which says it changes the list in place. `483_w_and_wf_forms_in_conditional_code` proves that a W condition written with cursor words such as `@NextNumber` and the same one written with raw index arithmetic such as `This[@i+1]` give the same positions.

None of the guards named here covers `Walkables`, `WalkNSteps` or `WalkBackward`; those lines rest on the runs kept for this chapter.

Next, what the field does better than the walker.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in laziness. A generator produces the next item on demand and can be endless. The walker builds a map of its whole journey the moment it is made. That is a win for foresight: you can ask which positions it will skip before it moves. It is a loss for a large or endless traversal.

The field leads in openness. A lambda carries any logic. The W language is bounded on purpose, and the price of the bound is the second path, WF, which has full host power and is meant for input you trust. There is also a quieter cost I measured once, on one case: a condition that calls something the language does not know gives an empty list and no error.

The field leads in reach and in tooling. Iterators and comprehensions are in every mainstream language, with editor support, type checking and a literature. These four metaphors exist in one library.

And I will not print the library's own comparison of walkers with loops and iterators. Its table has thirteen rows, all of them in the walker's favour, none of them run. A table is worth printing when its rows have been run and a loss stands beside the wins; for this one, that is the next step.

Next, the limit, in the stage words built, designed and absent.

7. THE LIMIT

BUILT	the one-dimensional walker, the checker, the yielder and the performer on lists; the condition language compiled by the engine, with its host-code twin
DESIGNED	the four metaphors as native engine modules (only the yielder is one); a walker over any indexed structure
ABSENT	runnable code for the three models of the article 'Bridging Minds and Code'; the edge table of the walker narration as written

What is built: the one-dimensional walker, in the max layer, and the walks on lists; the checker as `CheckW`, `ContainsW` and `FindW`; the yielder as a class backed by the engine, and as `YieldW`; the performer as `PerformW`; and the W condition language compiled by the engine.

What is designed: the metaphors as native engine modules. The engine's design marks all four as planned; only the yielder has an engine file. And a walker that is not about lists at all but about any indexed structure.

Six things are absent, or still to be aligned with their articles, at this commit, from my runs.

- The walking-at-the-edge article's table and the library differ in two cases; aligning them is the next step. A single step larger than the range raises where the table promises silence, and a mixed-direction path raises where it promises

acceptance. The two principles hold in the cases shown above. The book states the principle and prints only the cases that run; the table is not printed.

- The "Beyond Loops" article's code differs in places. The history of a walker is a list of walks; a backward walk takes no argument now; the direction prints as "backward" where the article says "reverse". The site's own run of the article kept ten of eighteen blocks.
- The three models of "Bridging Minds and Code": none of its code runs as written; in the site's run twelve blocks raised and one differed. So I told them in words.
- The mental-model article's getter `EmptyStrings()` is absent.
- On the dedicated checker class, `ContainsW` is in development: it raises today. The string walker's backward-with-condition walk is absent.
- And "no eval" is still on its way to everything. The near-natural resolver, the other direction of chapter 13, still evaluates host code. So do `StringsW` and `ItemsWhere` on the list, and `ContainsW` on the dedicated checker class: each still calls `eval` on a condition string. What I said above, that the engine reads a W condition and cannot run it as code, holds for the engine's path; for these three it is the next step.

Next come the six doors, from the programmer to the agent.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Ask the same four questions of any object, and when you reach for a loop, name the move: walk, check, yield or perform.
The analyst	Say the problem in plain words, and the verbs you used are the steps: "are all of them", "which ones", "give me the big ones", "make them negative".
The designer	The yielder never changes the list and the performer does, and the verb tells you which.
The decision-maker	A W condition is a string the engine reads and cannot run as code, so it can be stored, shown and approved. The price is a small language, and a call it does not know gives an empty list, not an error.
The educator	The self-study book already has the order: find, then apply, then walk, ask, produce and act.
The agent	Write the condition as the string, not as code. When the answer is empty, check the condition's words before you trust the emptiness.

9. EXERCISE

Take the list `[ 4, 9, 12, 7, 20, 3 ]`. Four sentences are given. One: are all of them above 2? Two: give me, as a new list, those above 8, doubled. Three: make the ones above 8 negative, in the list itself. Four: which positions are 2, 4 and 6?

For each, say first which move it is, walk, check, yield or perform, and write under the line you will write what you expect it to print. Then write the line, with the condition as a string wherever there is a condition, and run it. After the second and the third, print the list again. Say, before you look, in which of the two the list has changed.

I left this one in your hands, on purpose. Here is the block of section 3 with one change: the list is the exercise's list, and I print nothing under it.

HARO

```
o1 = new stzList([ 4, 9, 12, 7, 20, 3 ])
```

```
? @@( o1.FindW('{ @item > 5 and print("hacked") }') )
```

✎ not run: an exercise, written for you to run

Run it. The condition begins by asking whether an item is above 5. Look at what comes back, and at what does not, and say why before you go back to the sections. Section 3 says what the engine does with a condition that calls code; section 6 says what that costs, and section 8 says what an AI agent should do about it. When you have solved it, you will know why an empty answer is not yet a proof that nothing matched.

The next chapter turns to the writing style, and to the seven goals. Its principle is one sentence: what you think is what you write.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	learn
ON THE SITE	craft
RULINGS OF THE STUDY	12.3, 12.6, 12.14, 12.16, 12.17

PART II · CHAPTER 7

The writing style, and the seven goals

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the rest of the writing style, as a grammar of fourteen rules. I print the state each rule is in at the commit this book runs, a block where it runs and a limit where it does not. Under the grammar I put the seven goals, and I show three of them in blocks and leave the four I have not run to the study. I print the house court, the fourteen rules the library holds over its own code. I say what it does not read: the code of a program written with the library. The principle is chapter 5's, and I do not argue it again: what you think is what you write.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The word *style* means two things in the field, and I have to say which one I mean.

The first is style as layout, enforced by a tool. Python's style guide dates from 2001, Go's formatter from 2009, and every project runs a linter before a commit. They judge the shape of a line: spaces, capitals, length, unused names. They are good at it, and they run by default. They say nothing about whether a line follows the order in which its writer reasoned.

The second is the small language inside the language. Programmers already write small languages in strings, and nobody calls that a style: a regular expression, a SQL text, a format string such as `"%5.2f"`. Each is a separate notation, with its own tool, learned apart.

For goals, the field says *quality attributes*. The ISO standard for software quality, of 2011, lists eight of them, and architecture methods rank them before a system is drawn; that is general knowledge, not a study of the record. The seven goals of this chapter are a different list, chosen for a different purpose. They do not say what a product promises its users. They say what a feature must serve to be admitted into one library.

2. THE FIRST PRINCIPLE

What you think is what you write. It is chapter 5's sentence, from the diagram of 2020, and I do not argue it again. Chapter 5 took five rules of the style. This chapter takes the rest, as a grammar, and then the goals under which the grammar was designed.

For the goals, the point is mine, and not a dated quotation. A software architecture that is goal-oriented is the only way one architect masters a project of this size. I hold this as a view, not a result. What stands behind it on this page is one case: the 2022 matrix served as a skeleton for one architect over about three hundred thousand hand-written lines (section 6).

3. THE SOFTANZA FORM

My study of the record wrote the style down as fourteen rules, from reading the code. They are readings of the code, stated as rules, and not a standard anyone signed. Here they are with the state each is in at the commit this book runs. I put the table first because it shows which rules carry a block and which carry a limit.

#	RULE	STATE AT THIS COMMIT
1	A value is lifted to its object by `Q()` or a typed lifter	runs; chapter 5
2	A scene is a brace block on one object	runs; chapter 5
3	Every printed line carries its answer under `#-->`	judged by a machine; chapter 5
4	A parameter is named, and the name reads as English	runs; chapter 5, and here in any order

#	RULE	STATE AT THIS COMMIT
5	A chain mirrors the order of thought, one step per line	runs; chapter 5
6	The verb's form says what happens	runs; chapters 5 and 8
7	Suffixes and prefixes are small and mean one thing each	in part: `CS`, `XT`, `viz` run; the deck's `FF`, `dft` and `inf` families are not defined
8	A function has alternatives for the way a person says it	in part
9	Precision over generality, when it costs nothing	in part
10	Code travels inside an argument as a small string	the seat runs; the `@` on a parameter's name is retired
11	Small literal languages live in strings	runs in one form; the braced form raises
12	A sentence can be near-natural	not this chapter's; none of the deck's three forms ran as written
13	A function may be named in any human language	absent; section 7
14	A comment speaks before the code, in the teacher's voice	runs

Rule 11 first, because it is the strangest. The host language counts in integers and cuts the decimals. What does a range written inside a string keep? The comments below speak before the code, in the teacher's voice, which is rule 14. Run it:

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" ]
```

✓ ran: printed what this block promises

The host answers `[ 1, 2, 3 ]` to `1.02 : 3.08` and cuts the decimals. The string answers `[ 1.02, 1.03, 1.04, 1.05 ]`, and the two Arabic bounds give six letters! Softanza reads a small range language inside the string and keeps the decimals, the numbered words and the letters of any script. The range language is an argument like any other. Does it give the replacements for each occurrence, in order, and does the extended form wrap the palette when the occurrences outnumber it?

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
```

✓ ran: printed what this block promises

The extended form wrapped the palette, and ♥1 came back at the end!

A list can also be carried as a string in its short form, recognised, converted between forms and evaluated back:

HARO

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

✓ ran: printed what this block promises

Rule 10: code travels inside an argument as a small string with a placeholder, `@char` for the characters of a string and `@item` for the items of a list. Chapter 6 showed the seat on a list. Here it is on a string, where it finds and removes by a rule and not by a loop:

HARO

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

✓ ran: printed what this block promises

The deck of 2022 writes the same call with a named seat, `:Where = '...'`, and that form raises on a string in this commit. I do not show it. The same deck puts an `@` at the end of a parameter's name, `:With@`, to mark that its value is code to be evaluated. That form is retired. A guard's header says so: the eval-based forms are retired. The deck itself marks the form "TODO". Code now travels in the seat as a bounded string that the engine evaluates, or as a function.

Rule 8: the same act may be said in two orders of words, with the container's name, or with the side spelled four ways, left or start, right or end; two of the four run below. Each pair is one act, with two definitions. The library's own narration calls the first with two arguments and it raises `less number of parameters`; the guards pass the starting position, and so does the block:

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" ]
```

✓ ran: printed what this block promises

The deck promised more. A family of five names ending in `CharsRemoved`, for removing characters at the ends, is named in the deck. Two are defined, `LeadingCharsRemoved` and `TrailingCharsRemoved`. Three are not: `LeadingAndTrailingCharsRemoved`, `TrailingAndLeadingCharsRemoved` and `RepeatedCharsRemoved`. `RemoveLeadingAndTrailingChars` is defined, and is not one of the five.

Rule 9: precision over generality. The precise name says what is counted. A string's characters are counted by `NumberOfChars`, and `Size()` is the host's byte length. The two list verbs in the same block show the rule on a list: the precise name, `ContainsBoth` or `ContainsAll`, says what is asked. Do the two answers differ on an Arabic word? Run it:

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
```

✓ ran: printed what this block promises

The run answers `4` to the one and `8` to the other. The word has four letters! The library's own documentation line for `NumberOfChars` lists *size* among its alternative words. The run shows that `Size()` does not mean it. I list it among the

requests.

Rule 7: a prefix `viz` shows the answer under the word, and the suffix `XT` adds the positions as numbers; chapter 8 runs the block, among the forms of a name.

The narration and the deck write `vizFindAll` on a string. It does not exist there at this commit: my run answered `Calling Method without definition: vizfindall`. `vizFind` does exist. A list has its own `VizFindAll`.

THE GOALS

Under the grammar there are seven goals, written in the 2022 deck as a matrix of thirty-three features, which section 4 counts. My study reads them as the library's architectural skeleton: a feature that serves none is refused, and a goal with no feature is a gap. The study's summary is a session's reading and not a text of the library, and no document in the library states the rule in these words. The nearest is the engine's design document: the engine must enable all seven.

I will not list the seven as a table of slogans. Here is what runs of three of them.

Flexibility: the order of named parameters is free. Are two calls, with the parameters in two orders, the same call?

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()
#--> ----@@--@@-------@@----##---
```

✓ ran: printed what this block promises

Both orders print `----@@--@@-------@@----##---`! This is the free order of named parameters; the deck's other free order, a number and a string swapped in their positions, is chapter 8's, and it does not run. A tension must be told with it.

The design of Haro refuses free-order parameters as grammar, and the library keeps them. Two facts are true today. The block above runs them, so they are in the library. The grammar a machine is held to leaves them out, by the Haro principle. Whether the library keeps them past that grammar or retires them is a decision the record does not hold. I do not make it here. They are kept, and they are not grammar.

Consistency: one vocabulary across a string and a list. Does the same verb answer on both?

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
```

✓ ran: printed what this block promises

The loss beside it, from the same slide group: the deck's advice to use the keyword Item with any enumerable type does not hold. `NumberOfItems()` on a string raises in my run: `Calling Method without definition: numberofitems`. On a list it answers.

Manageability: a failure that says where, what, why, and what to do. The library raises an error in four fields. I do not print its message, because in my run its first line carried a line number of the library, and a line number is not a promise.

The other four goals (expressiveness, reliability, the human metaphors and the practical abstractions) I leave to the tables of the study. I have not run them for this chapter, and I will not describe from the deck what I have not run.

Next comes the decision, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused a style that was changed when AI became the main writing tool. Chapter 5 tells what I found, and what I let go. The implementation code that the AI wrote is accepted as good. What is demanded is that every line a programmer, human or AI agent, is asked to write with Softanza embraces the historic style.

I refused a goal that lives only in a document. The seven goals are written in one design document and in tags that no court reads. I found no court over either.

I refused to hide the cost of the style. The study measured markers per thousand lines. The narrated guards, the test files written to be read as articles, carry the promise `#-->` in under one line per thousand. They carry a call to the assertion helper in eighty-five. The older numbered tests carry the promise in fifty-two. The library's code carries the promise in less than two. The style is densest where I wrote for a reader, and thinner where the AI wrote for others; carrying it there is the next step. A marker is a reading of the text, not a judgement of the code; the study says so itself.

I refused to recount the tags as mine. The forms of the functions are marked in the code by info-tags, in comments that begin with `#<` and an `@`, applied by hand. The study reads them as the feature-level vocabulary of two goals, expressiveness and flexibility, and counted 6,090. I counted again on 8 October 2026 and found 6,126, with 89 spellings. But 1,360 of them sit outside the archive folders, and 4,766 sit in the archive folders. So the sentence "applied by hand at 6,090 points" is, at this commit, mostly a sentence about the archive. No guard reads them: the reflection layer reads a method's name, with a grammar, and not its tag.

I refused the tally as a count of passes. The study counts thirty-three features: twenty-four built, four transformed with the project, two that changed form, one partial, one archived, one a name only. The arithmetic holds. But it is a count of names found, and a definition found is not a passing test. Several of the features are named in the deck and not defined in the library's code, the archive aside. They are the families `FF` and `FX` for sections, the families `dft` and `inf` for defaults and infinities, `WalkFF` and `AddWalkerFF`, and the `Check` on a list. That last one is the deck's checker, which answers 1 or 0 for each item. Two are defined and do not yet match some of the deck's examples. `Yield` should give the result of an expression for each item. `Perform` should apply an action to each item. I print the tally, and I print its losses with it. The deck also says "7 root causes of programming complexity", beside the seven goals. No slide lists those seven causes, and no other file in the record names them. The chapter names none and does not match them to the goals one by one.

The court comes next, and I run it.

5. THE COURT

`coderule_narrated` proves that a house rule is a rule over the code graph; that planted code is found by name; that clean code passes; and that the library lists fourteen. `testoor_promises_narrated` proves that the promise `#-->` is judged as a claim of a stop: kept, diverged, unreached or unjudged. `602_l_function` and `59_use_of_the_l_small_function` prove the range language. `527_identifying_lists_inside_a_string` proves the lists hosted in strings. `48_replacebymany` proves that the replacements go in order. `483_w_and_wf_forms_in_conditional_code` proves the condition in a string, in two spellings of the same scan. `28_numberofoccurrenceofcharleftside` proves the alternative names in Arabic as well as Latin, and carries a note that the archive's promise was wrong and the implementation is the authority. `646_content` proves the named parameters. `972_narration_viz-finding_a_recurring_substring` proves the viz tour.

Here is the house court itself, the fourteen rules of the library and not the fourteen of the style. The numerical coincidence is a coincidence. What does it answer to a planted misspelling, and to a planted write to a constant? Run it:

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
```

✓ ran: printed what this block promises

The misspelling is found by name, `misspelled_name`, and the code stays clean, `TRUE`, because it is a warning. The write to a constant is found by name, `writes_a_mutable_constant`, and the code is refused, `FALSE`, because it is an error. The court refused the write to a constant!

A detail of the spelling, for Appendix B: the names list uses hyphens, and the finding of eight of the fourteen comes back with underscores. The block shows it: the list prints `misspelled-name`, and the finding prints `misspelled_name`.

What the court does not read is the code of a program written with the library. The fourteen rules of the house are rules of the library's own code; a rule of writing for the code written with it is designed (section 7).

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in enforcement by default. A formatter runs on every commit. For the code a programmer writes with Softanza, the court that would enforce the style is designed and not yet built, and chapter 5 says so.

The field leads in having its goals somewhere a reader looks first. The seven goals are in one design document and in tags that no court reads, and the operating notes of the library's own repository name none of them. An AI reader's account of the library says that sixty paradigm documents could easily be sixty disconnected ideas. The study reads it as a sign that the discipline is in the code and written nowhere a reader would look first. It is a session's document, and not an independent witness.

The site once showed a reduced dialect: its blocks had no named parameter and no scene. It has since put the fourteen rules on its Craft page and the rules as data. I did not re-measure the site's blocks after that.

Where I think the style stands apart is that every rule serves a step of thought, and every step has a written form. The verb's form is a contract on the object. And the promise under the line is the example, the test and the documentation at once. The 2022 matrix served as a skeleton for one architect over about three hundred thousand hand-written lines, by my study's count.

The limit, in the next section, says what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	the library's fourteen house rules as a court over its own code; the promise judged as a claim; the small languages of ranges, lists and conditions in strings; the code seat; the viz prefix; alternatives and named parameters in any order, in part
DESIGNED	a court that judges the style of a program written with the library; a court that reads the goals; the writing grammar declared in Haro; the classic form of a test block as the one form

ABSENT — any machine that judges the style of the code you write today; the writing in any human language of a function's name; the near-natural sentence as the deck wrote it; the eval-based code on a parameter's name

What is built: the fourteen house rules of the library as a court over its own code, run in the block above. Also built: the promise judged by a machine; the small languages, the seat, the prefix. And the features that run (`CS`, `XT` and `viz`, named parameters, the same verb on a string and a list; `ZZ` runs in chapter 8).

What is designed:

- **A court that judges the style of a program written with the library.** It is a specification in the study: the fourteen rules as house rules over a snippet's graph. Its examples are a positional argument where a named one exists, or a printed line with no promise. I searched the library for it and it is not built yet; it is the next step.
- **A court that reads the goals.** It would read the tags. To come.
- **The classic form of the test block as the one form for new blocks.** The study's ruling is that the revival "is not a rewrite" but a decision. The classic form is the one form for new blocks, and the converted blocks are left as they are. The test tree's own README still calls the classic form the authoritative test surface. I found no ruling that makes it the one form, so it stays a decision, and chapter 5 names it.
- **The writing grammar declared in Haro.** The vocabulary half exists: the forms of the library's names are data in Haro, and the session's record is 584 of 595 shipped spellings agreeing. The usage half, the scene, the chain, the code seat and the promise, is not declared.
- **The discipline declared once.** Partly done on the site, as data, which is the page's source and not a court.

What is absent:

- Any machine that judges, today, the style of a program written with the library; beside it, the house court judges the library's own code.

- Rules 12 and 13 as running forms. In my runs, the deck's near-natural samples did not run as written, and neither did a function named in Chinese or in French.
- The `@` on a parameter's name as a working form.
- The braced form of the range language, `L('{ "♥1" : "♥6" }')`, which raises, and ranges that change width, such as `day8 : day11`, which also raise, with an uninitialised-variable error from the host. The list constructor from a string raises too. These three come from my runs; no block of this chapter keeps them.
- The visualised call stack, and the cache and log decorators, which exist as names in comments.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Fourteen rules you can hold your own line against. Lift the value, ask in a scene, put the answer under `#-->`, name the parameter, chain one step per line, read the verb's form; and put a range, a condition or a palette in a string.
The analyst	The words of your problem become the names of the steps. A rule written in a string is a rule you can store and read, and not code you must trust.
The designer	The viz prefix draws the answer where the eye expects it. The goals are the headings under which a design is judged, one feature at a time.
The decision-maker	Seven goals and thirty-three features are a reading grid. In the study's reading, a feature that serves none is refused, and a goal with no feature is a gap. The tally is printed with its losses.
The educator	The discipline is taught whole and in order. The converted guard for lists in strings keeps the old promise's wrong spacing in a comment, which is a lesson on the cost of hand-typed answers.
The agent	The rules are given to you as data on the site. For the code you write, the court is designed and not yet built. Write the question, then the promise under it, and put your condition in the seat, not in an evaluation of a string.

9. EXERCISE

Write, before you run anything, the line each task needs and the promise under it:

1. From one string, the list `[ "♥7", "♥8", "♥9" ]`.
2. The string `"haro php haro"` with the two `haro` numbered `♥1` and `♥2`, in one call that uses the small language of task 1.
3. The list `[ "C", "B", "A" ]` ordered by swapping the first and the third items, with the positions named.
4. The count of characters of `"سلام"`, with the name that says what is counted.
5. A guess: what does `L("day8 : day11")` print?

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

✎ not run: an exercise, written for you to run

Run it. For the fifth, you were allowed to be wrong, and you were probably wrong: the range changes width, so `L` hands its text to the host to evaluate, the host reads `day8` as a variable, and it raises with `Using uninitialized variable: day8`. That is the limit of the small language, learned by being surprised once. Do not change your promise to match the output. Explain the output first.

I left this one in your hands. The first call of the rule 8 block, with its named start taken out, has two arguments. Nothing else changes:

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

✎ not run: an exercise, written for you to run

Run this one and read what comes back. Then repair the call. Section 3 says where this form of the call comes from and what the guards do instead. When it is repaired, you will know why a call that the library's own narration writes raises, and what the guards pass that it does not.

Chapter 8 takes a function's name whole, as a linguistic expression, and then scope: a function's name is a sentence, and its prefixes, suffixes and parameters are a grammar that the library can read.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	craft
ON THE SITE	principles
RULINGS OF THE STUDY	12.2, 12.14, 12.16, 12.17, 12.21

PART II · CHAPTER 8

Functions as linguistic expressions, and scope

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 13 of 13 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I read a function's name the way the library reads it: a sentence whose prefixes, suffixes and parameters are a grammar. I walk the forms of a name that run, and I count the five that do not. Then I turn to scope: the same verb, three verdicts, with the frame in the verb and not in a flag set earlier. I say what Haro's design keeps of the forms, what it refuses, and what the court proves. The principle is this: a function's name is a sentence, and its prefixes, suffixes and parameters are a grammar that the library can read.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In the field, a function's name is a label. It follows a convention, camel case or snake case, and it tells you roughly what the function does; the details are in the signature and the documentation. A name does not carry grammar. The related tools are all about structure. Python and Swift offer the keyword argument, so that a call can say which parameter is which. The fluent interface chains calls so that a line reads like a sentence, and the overload gives one name several signatures.

And when the field wants a function to behave differently in a different context, it changes its meaning by a flag set somewhere else. The flag is a mode, a locale, an environment variable. The function's name does not tell you which frame it will run in.

2. THE FIRST PRINCIPLE

A function's name is a sentence, and its prefixes, suffixes and parameters are a grammar that the library can read. The paper that first made the claim says, in the words the study of its roots quotes, that it was achieved "without machine learning or NLP systems". It was done by naming conventions that encode meaning, a parameter order that mirrors speech, flexible forms and intelligent defaults.

3. THE SOFTANZA FORM

Look at a name the way the library looks at it. `FindNextSTCS`: find next, from a start position, with a case flag. `RemoveManyCS`: remove, many things, a case flag. Each suffix is small and means one thing, and, in these names, the order of the suffixes is the order of the parameters. Here the first name finds. What does the second leave when it removes every occurrence of three letters, whatever their case? Run it:

HARO

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

✓ ran: printed what this block promises

The article promised `SOFTANZA` on the second line. The run prints `STAZA`! The code is right and the article is wrong: a removal that does not care about case removes the capitals too. I print the run.

Does the library only follow this grammar, or does it read it? Its name parser, `_StzParseName`, is an internal helper that no guard covers; I call it first below, for the raw reading. The public way in is `ExplainMethod`, which explains a method in those terms:

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
```

✓ ran: printed what this block promises

It reads it. That is the claim, made mechanical: the grammar is in the code and not only in the article. The printed line `reads as: remove -- over many values at once` is the parser's reading of `RemoveMany`. The parser decodes the prefixes and the suffixes, the words "except" and "is not", and the voice: a name that ends in `ed` is passive.

I will walk the forms now. Chapter 5 showed four of them: the active, the passive, the fluent and the named parameter. Here are the others that run.

Z and ZZ say whether a position comes as a number or as a section. A suffix, and the answer changes shape:

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 ] ]
```

✓ ran: printed what this block promises

The prefix viz changes the kind of answer: a drawing. The position is shown under the text, and an option can number it:

HARO

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

✓ ran: printed what this block promises

The plural extends a verb to many things without a loop, and **the exceptional form** keeps only what you name. The article's example for the second, `RemoveNonLettersExcept`, does not exist. The form does, in other names:

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
```

✓ ran: printed what this block promises

The negative is a word in the name, not a NOT around the call:

HARO

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

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

✓ ran: printed what this block promises

The deep form answers a path, a list of indices that goes as deep as the data does, and the same family replaces and removes at that depth, with `:By` as its named seat where chapter 5's `Replaced` said `:With`; the deep verbs accept both:

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", "♥" ]
}
```

✓ ran: printed what this block promises

The immutable form, QC, runs the whole chain on a copy. After that chain, what does the original hold?

HARO

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

✓ ran: printed what this block promises

The chain printed `H A R O`, and the original still holds `haxxrxo`!

The misspelled form is accepted, with a note on what stands behind it. A misspelling of a name works:

HARO

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

✓ ran: printed what this block promises

The article says the library recognises and corrects minor typos, and that the intelligence resides in the library itself. The code shows how the form works: `WithoutSapces` works because that spelling was written by hand as an alias. Another misspelling, `WithoutSpces`, fails. So the form is real, and in what I traced it is a list of declared aliases; the correction engine the article describes is not among what I traced. I did not trace every file that mentions correction; one message in the object class only suggests a name, with "Did you mean".

Now the scope half of the chapter. The same fault shows up everywhere in programming: behaviour with an invisible governing frame. I hold this as a view, not a result; what stands behind it here is three instances: the regex verbs, the system and the placement of a deployment. A function does one thing here and another there, and nothing in its name says which frame it is in. The answer, in five moves, is the paradigm I call scope-oriented programming. Four come first. Name the hidden frame. Turn it into a small, closed set of named scopes. Choose the scope at the call site, in the verb, and not in a flag set earlier. And give the library the scope to reason with, a semantic handle and not a comment.

It was born in regular expressions, where the frame is how much of the text a pattern must cover. The same pattern, the same text, three verbs: what does each answer?

HARO

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

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

✓ ran: printed what this block promises

`Match` answers `FALSE`: it asks whether the pattern covers the whole value. `MatchFirst` answers `FALSE` too: it asks whether the pattern occurs anywhere, with the line anchors off. `MatchLine` answers `TRUE`: it makes `^` and `$` bind at each line. The frame is in the verb. There is a word scope too, which supplies the word boundaries so that you do not write them. It rewrites the pattern of its object for good, which is why the second instance is a new object with the same pattern. Does the same pattern answer alike under `MatchFirst` and under `MatchWord`?

HARO

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

✓ ran: printed what this block promises

The same pattern answers `TRUE` under `MatchFirst` and `FALSE` under `MatchWord`!

The second instance is the system. Feature code is written in a named scope, and the same verb gets a different verdict in each. The scope can make a laptop behave like a device it does not have, or refuse what the target forbids. Here, one solution with a backend on a Linux server and a firmware on a small board. What does each system answer when the same verb, `Spawn`, is asked of it? Run it:

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.
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The same verb, `Spawn`, has two verdicts! The server answers `native`; the board answers `scope 'esp32' (espidf) forbids 'process'`. The scope decides, not the operation. And the laptop has no pin to read, so the scope answers `rehearsed`. This is the fifth move: when a scope's capabilities differ from the host's, the library bridges the difference, in both directions. Two declared systems can be compared as data, with a closed set of capabilities:

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" ]
```

✓ ran: printed what this block promises

The run printed `[ "filesystem", "process" ]` for what the laptop can do that the board cannot, and `[ "gpio" ]` for what the board needs that the laptop lacks. That is the second instance. The third is deployment: the target is the frame, and the scope decides not to ship what the target already does well; chapter 4 showed the plan. The fourth is concurrency, and it is a plan.

The design is next, with what Haro keeps of these forms and what it refuses.

4. THE DECISION, AND WHAT I REFUSED

In the Haro design, principle 7 makes forms a grammar that the language owns. It is designed and not built: the forms are declared once, and the compiler would generate and check each family. The passive form may not mutate; a verb with Q returns the object; suffix order is parameter order; a W takes a literal parsed at compile time. The spike judges names only: it does not read bodies and it does not judge suffix order. Its first measure is the library's string class: of its 595 shipped spellings of the verbs find, replace and remove, 584 agree with the declaration and 11 do not. I told in chapter 5 what Haro refused as grammar: the misspelled form, the free order of parameters and per-language method names. Here is what it means for this chapter: five forms do not run.

The future form was trimmed on purpose. The header of the natural-code file dates it 10 July 2026: the global queue of deferred actions left, and its FQ and FFQ wrappers were already gone. The reason is written in the design: natural language demands building the whole computational model before executing, and the engine's token plan now does that. It is a plan of the natural engine and no longer a suffix.

The conditional form changed its mechanism. The narration says it no longer goes through `eval()` but through a small text language that the engine compiles, and that the old extended form, WXT, is now a deprecated alias of W.

For scope, the design document refuses the floating current. Feature code is never written against an ambient "current system". It names its scope. And for concurrency the document records two refusals before there was any code. One is a method per operation that smears the frame across many verbs, such as `DoubleEachConcurrently`. The other is a mode set earlier. That mode is the antipattern this paradigm was founded against. It also refuses a scope that no host can perform. Its first guess for concurrency was thread vocabulary, and the language beneath has no threads.

I refused to ship the whole engine to the edge. The placement rule is: ship only what is critical and either unique to Softanza or weak on the target. A target that already excels at something gets nothing shipped.

Now the court, and which guards prove which of these claims.

5. THE COURT

`00_regex_narrated` proves the verbs `Match` and `MatchFirst`: `Match` anchors, `MatchFirst` searches, Z hands back positions and does not print them. It asserts nothing for `MatchLine` and `MatchWord`: the blocks above show them, and only classic blocks, which are not guards, cover them. `972_narration_viz-finding_a_recurring_substring` proves the drawn answer. `full_scope_model_narrated` proves the system scope, with its three verdicts. `system_scope_model_narrated` proves the closed set of capabilities; one of its assertions is a timing limit, and section 7 says what one probe showed of it. `delivery_narrated` proves the placement plan.

Where the field does better comes next.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in convention. A name that is a label is easier to teach and easier to type. The grammar of this chapter has to be learned, and some of its forms are rarely used.

The field leads in the guarantee. Python's keyword arguments are checked at the call, and Swift's labels by the compiler, for every function, from the first day. The forms here are judged by the Haro spike, on names and not on bodies, and by the library's own parser, and only in the names that follow them.

On scope, the field leads in maturity. The closest field practice is the explicit context: the Go habit of passing a context to every call, a locale object that carries a frame. They are older and more widely used. What this paradigm adds is that the

frame is one of a small, closed set, chosen in the verb, and the library can reason with it.

Last, the limit, with what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	fourteen of the twenty-two forms, some with examples other than the article's, and three more in another shape; the library reading a name as a sentence; scope in the regex verbs, in the system and in the placement of a deployment
DESIGNED	the forms declared once as data and generated, in Haro; the scope paradigm for time, units, memory and style
ABSENT	five forms: partial, future, free, default and multilingual; concurrency as a scope: a plan with no code

What is built: fourteen of the twenty-two forms of a name run, some with examples other than the article's, and three more run in another shape. The library reads a name as a sentence. And scope runs in the regex verbs, in the system, and in the placement of a deployment. The article counts twenty-three sections: the twenty-two forms of a name, and "small functions", which are functions and not a form of a name. The book's number is twenty-two, the forms of a name.

What is designed: Haro's forms as data, with generated families; and the scope paradigm for time, units, memory and style, as rows of a table.

What is absent, and the page should say each with what exists beside it:

- **Partial**, `@Removed`, is in the archive only: the call is not defined.
- **Future**, `BeforeQ`, was trimmed on purpose, as I said.
- **Free**, the suffix that lets a call give its parameters in any order and any subset: there is no definition of it outside the archive. The library's own parser knows the name `FX` and not `FF`. The two narrations that tell it use two names, and the code has neither.

- **Default**, `dftRange` and `infRange`: not defined; the library's comment calls the informing form not yet built.
- **Multilingual**, a method named in French or Arabic: not in the live library. The names exist in the archive. The nearest thing today is the language packs of chapter 11.
- The three that run in another shape. The exceptional form exists, but the article's example does not. Free order, as the article means it, a number and a string swapped in their positions, works for the first order only: in my probe of 8 October, the swapped call ended the program with a status of one and printed nothing. The free order of named parameters is another thing, and chapter 7 runs it. And the statement form is half in place: the "All" half works, and the "No" half is the next step. The library's own classic test promises true on two of its four "No" lines, and the run prints false on all four.

On the scope side: the regex verbs differ from the table of the scope design document in two places. `MatchLine` lets a dot cross a line, where the table says it stays within one, and `MatchOneSegment`, which the table offers for the lazy case, is not defined. The system scope is built, and the design document still calls it designed: the record contradicts itself. The scope design document, version 1.2 of 16 August 2026, is quoted here as the document's text. The record does not say who wrote it, and I do not print it as mine. The system foundation document of 20 July 2026 says the foundation is implemented. In one probe run under load, which the book's data does not keep, 61 of the 62 assertions of its narrated guard passed, and the one assertion that did not hold was a timing limit. I state what the code does: the system scope is built, with that result printed beside it, and the design document's word "designed" is the document's, out of date. And concurrency is a plan of record with no code: the word `Concurrently` has no definition.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Read a name as a sentence: its suffixes are its parameters, in order.
The analyst	A position can come as a number or as a section; a name says which.
The designer	The prefix changes the kind of answer: viz draws it.
The decision-maker	A scope is chosen in the verb, so the frame is visible where the call is made, and a review can see it.
The educator	The library parses a name into its sentence, so a learner can ask it what a method says.
The agent	Write the scope in the verb. A call that names a scope is not moved to another frame by a flag set elsewhere.

9. EXERCISE

Take one verb from a library you use that behaves differently in two frames: a mode, a locale, a target. Say in one sentence what the hidden frame is. Then write, as you would like it to be, two names for the same act in the two frames, with the frame in the verb. Before you run anything in Softanza, take `Match` and `MatchLine` on a text of three lines and write what each prints for a pattern that anchors at the start.

Run it. If your two names differ only by a flag, you have not yet moved the frame into the verb.

I left this one in your hands, on purpose. In section 3 the article promised `SOFTANZA` from a removal, and the run printed `STAZA`. Here is that block with one change, the last argument of the call:

HARO

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

✎ not run: an exercise, written for you to run

Run it. Set what it prints beside what the article promised and beside what the first run printed. Section 3 says why the first run did not give the article's line; I do not print what this one gives. When you have solved it, you will know what the case flag decides in a removal, and whether it alone can keep the article's promise.

Chapter 9 turns to patterns of thought. It presents the Softanzuter, a computational representation of a thinking machine. It is based on one or many Regexuters, which identify and react to patterns of thought. With it come Reaxis and the engines.

> WHERE THE EVIDENCE LIVES
>
> | | |
> |---|---|
> | NARRATION | `stz-functions-as-linguistic-expressions.md` |
> | NARRATION | `stz-function-forms-narration.md` |
> | NARRATION | `stz-system-dev-to-deploy-narration.md` |
> | ON THE SITE | craft |
> | ON THE SITE | innovations |
> | RULINGS OF THE STUDY | 12.3, 12.13, 12.19, 12.21 |

PART II · CHAPTER 9

Patterns of thought: the Softanzuter, Reaxis and the engines

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 8 of 8 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the three subjects of the title in turn: the Softanzuter, Reaxis, and the engines, of which the timeline and the grid run here. The Regexuter binds a pattern to code that fires and remembers why; Reaxis is reactive programming with its vocabulary redesigned. I climb the ladder the architecture draws, four rungs, and at the third I run the block that fails to cascade rather than claim a rung that is not built. I say what I refused: regex as a way to read text, the field's words for reactive programming, the four siblings, and the cascade. Everything serves one sentence: a Softanzuter is a computational representation of a thinking machine, based on one or many Regexuters, enabling it to identify and react to patterns of thought.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field has three tools for making a program react. Regular expressions recognise a shape in text, and nothing more: they say "this matches", and they stop. Reactive programming, the family of observables, subscribers and schedulers that Rx made popular, lets data flow through a program over time. Its vocabulary comes from plumbing, telephony and publishing: pipes, backpressure, hot and cold sources. And rule engines, the production systems of the field with their

forward chaining, fire rules when facts appear, and keep firing while new facts appear; that is general knowledge, and the rule engines of the field are older and more mature than anything in this chapter.

Each tool is separate. A regular expression does not trigger a computation, a reactive stream does not recognise a pattern, and a rule engine does not have a vocabulary that a person would use to say a thought. I hold this as a view, not a result.

The next section gives the one sentence I answer with.

2. THE FIRST PRINCIPLE

I gave this paradigm one sentence, and everything below tries to make it mechanical: **a Softanzuter is a computational representation of a thinking machine, based on one or many Regexuters, enabling it to identify and react to patterns of thought.**

The next section shows how far the library has made it mechanical.

3. THE SOFTANZA FORM

The architecture draws it as a ladder with four rungs, each adding one thing.

The first rung is the pattern. You will see it in chapter 15: a shape in a medium, a string, a list, a number, that can say "this matches, and here is how", and do nothing more. The second rung gives a pattern a reaction: the Regexuter, a reflex arc. You bind a pattern to code, process a text, and the code fires on every match. A Regexuter reacts, and it remembers what it did. Here it redacts the secrets in a sentence. What does the state it keeps say about each one: what it matched, what it computed, where, and which trigger fired?

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" ]
}
```

✓ ran: printed what this block promises

The state is provenance: every reaction that changes a value keeps its why. The last line, `[ "sensitivedata" ]`, is also an honest limit, and I will come back to it: what the Regexuter records as a dependency is today only its own trigger.

Now the third rung, the one the definition is about. It is many Regexuters, over different media, sharing one state, so that what one faculty writes is a medium another faculty can match over. A thought is a state entry with its provenance, and a settled thought is a state in which nothing more fires. That rung is not built, and the block shows how I know. The second trigger below is written for numbers of three digits or more. The first trigger turns the 7 into 700, which is one. If the machine thought in cascades, would the second fire on the first's result? Run it:

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" ]
}
```

✓ ran: printed what this block promises

It did not: the results read `[ 700 ]`, the triggers that fired read `[ "number" ]`. `BIG` is nowhere! Every trigger is matched once against the original text, and what one trigger writes is not offered to the others. That is the whole difference between the second rung and the third.

The cascade runs one rung above, at the fourth rung, the AI agent: the thinking machine given a body in a world, and a goal. Its mind is made of skills, each with a condition and an action, which chain. The second skill fires on the fact the first one wrote. `Pursue()` returns how many skill firings were verified, and goes on until the world stops changing, or for twenty rounds at most. What does it answer when asked again?

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
```

✓ ran: printed what this block promises

It answers `2`, then `0`, and between the two the fact `served` reads `TRUE`. That is a fixpoint. The guard that proves it labels the scene in its own words: `the cascade to fixpoint (Softanzuter at agent level)`. In chapter 17 you will see an agent of this same class refused at a gate. Here is the other face of it: the agent that chains, because its mind is programmatic.

So the word has three meanings in the record, and the reader deserves to be told which is which. First, in this chapter, "Softanzuter" is the third rung, the thinking machine of the definition: designed, not built. Second, the engine has a module with the same name, which the macroplan presents as the foundation for AI agents. Its slots hold a state and a bounded mailbox, and a message into a full queue is refused, counted and never dropped quietly; the agent of the block above, `stzPIAgent`, never calls it. Third, the design of a universal engine behind all of it, a medium where any pattern language becomes a computation, is planned.

Reaxis is the second subject of the chapter's title, and it is a different idea. It is reactive programming with a vocabulary redesigned from the ground. A reactive thing is a Container, a Stream and an Rfunction: a function in waiting, which is not a callback. Data flows through a stream in plain words, a transformation, a filter, what to do with what passes. Four prices go in, 80, 90, 100 and 120: which ones come out, and at what value?

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 ]
```

✓ ran: printed what this block promises

`aSeen` reads `[ 108, 120, 144 ]`: the 80 became 96 and was dropped by the filter; the 90 became 108 and passed. Three functions are in waiting, and nothing here is called a subscriber; the old word `Subscribe` survives as an alias of `OnPassed`.

Reaxis replaced backpressure with overflow, a word you do not need a diagram to understand. And it was audited. A reading of the four strategies the article documented found that three of them lost data and reported none. The default discarded everything past the capacity while the counter read zero. That is fixed, and proved for all four strategies. Here, four items go into a buffer of two. What does the stream say happened?

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
}
```

✓ ran: printed what this block promises

It says `currentBuffer` is `2` and `droppedCount` is `2`. The counter that read zero now counts!

And for what the field calls debounce, Reaxis says what it means: wait for the attribute to settle. Three changes in quick succession: how many reactions, once the value has stopped changing for fifty milliseconds?

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()
```

✓ ran: printed what this block promises

One reaction. `nSettled` reads `1`.

The chapter's third subject is the engines, the small paradigms the library rethought one by one. Chapter 3 gave you truth and emptiness. Two more, run here. Time is a workspace, not a number. A timeline holds spans and computes the gaps and the overlaps. What does it say of the gap between two phases, and what does it do with a point that would fall into a blocked region?

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
}
```

✓ ran: printed what this block promises

The gap is named by what it lies between, `PHASE1` and `PHASE2`. The blocked span refused `EVENT2`! The refusal carries its reason: `Point 'EVENT2' falls within a blocked span or blocked point`.

And a grid has a cursor that knows where it is: position is column and row, and moving past an edge stays at the edge. Two moves down and three to the right, then three more to the right: where is the cursor?

HARO

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

✓ ran: printed what this block promises

At `[ 4, 3 ]`, then at `[ 5, 3 ]`. Three more moves to the right gained one column, and the cursor stayed at the edge!

4. THE DECISION, AND WHAT I REFUSED

I refused to leave regex as a way to read text. I made it a computational medium. A trigger fires a computation, the computation writes a state when it changes the value, and the state records its why. The Regexuter is a faculty and not a mind, and it holds no goal. I added the goal two rungs above it, at the fourth rung, in the agent, on purpose.

I refused the field's vocabulary for reactive programming. Observables, subscribers, backpressure, hot and cold sources, schedulers: borrowed from plumbing, telephony and publishing, and not from the problem. A function in waiting is an Rfunction. Data flows, and nothing is called a producer or a consumer. Overflow replaces backpressure. Waiting for an attribute to settle replaces debounce. The old spellings still work as aliases: feed, emit, send and receive all call one method, so my change was one of naming and not of removal.

I refused to build the siblings. The article that tells the Regexuter describes four more: an analyser, a genetic one, a linguistic one and a quantic one. In the genetic one patterns evolve against a fitness, in the linguistic one triggers are grammar rules, and in the quantic one a pattern can be in several states at once. The article gives each a sample call, and no class stands behind it. The design later found a purpose for two of them. Answering by example demands pattern induction, the genetic one's ground; several acceptable answers demand superposed candidates, collapsed by rules or by choice, the quantic one's. Chapter 18 tells where that stands. They remain a vision; two have their purpose written.

I refused to claim the cascade. The library's architecture says it itself: the scaffolding anticipates cascading but does not yet re-fire dependent triggers to a fixpoint. I would rather show you the block that fails to cascade than a block that does.

5. THE COURT

The Regexuter and the Listexuter have classic tests, and no narrated guard. The Regexuter's eight classic tests ran without error: the folder's log reads `PASS 8 ERR 0`. Nothing there checks what they print. Five of the Listexuter's seven fail their own expected output, and two promise nothing. The agent's cascade is proved by `piagent_narrated`, scene five. Reaxis has many guards. `00_overflow_narrated` proves the four overflow strategies and the case that loses nothing; `60_reaxis_on_reactor_narrated` proves a one-shot timer, the repeating timer, non-blocking HTTP and the settling attribute on the engine's loop. The timeline has `test_timeline_regression`, which covers construction, duration, adding points and moving the ends, and not gaps, overlaps or blocked spans. The grid has `00_grid_narrated`, and deep operations `01_deeplist_narrated`.

The next section sets this record beside the field's.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field's rule engines lead. They are older and more mature than the Regexuter, they chain to a fixpoint, and the algorithms behind them are decades old. What I claim is not a rule engine. It is the condition side: rules that are sentences of the pattern languages, over the two media that have a reacting machine built, text for the Regexuter and lists for the Listexuter. The state keeps, for every firing that changes a value, its why. The library's own architecture calls the Regexuter a nascent forward-chaining rule engine, and I will not call it more.

The field's reactive libraries lead in reach. Rx exists in most mainstream languages, with a large set of operators; that is general knowledge. Reaxis is one library on one event loop. The engine's own module for it is a small table of substring rules: the Container, Stream and Rfunction model lives in the library, on the event loop of the engine.

I will not print the star-rating comparison with the field's tools that the library's article on reactive objects carries. It is undated and unsourced.

The next section says where all of this stops.

7. THE LIMIT

BUILT	the Regexuter and the Listexuter, reacting with a state that remembers; Reaxis, reactive programming in plain words, on the engine's event loop; the timeline, the grid and deep operations; the AI agent whose skills cascade to a fixpoint
DESIGNED	the Softanzuter proper, many faculties sharing one state and cascading until it settles; the universal engine behind it; its four siblings, analyser, genetic, linguistic and quantic
ABSENT	a dependency chain that links one trigger to another; a section-merging verb the article presents as its point

What is built: the Regexuter and the Listexuter; Reaxis on the engine's event loop; the timeline, the grid and deep operations; the AI agent whose skills chain.

What is designed: the third rung, many faculties on one shared state, cascading until it settles. The universal engine for it is planned. The four siblings are vision. The macroplan marks the engine module as done, and the architecture table marks it planned: the same name for two things. That table marks every module of its fifth layer planned, including ones that have source, so it is stale on this point. The universal medium is planned in the engine design document.

What is absent or broken at this commit:

- The dependency chain is not one. Each state entry depends only on its own trigger, so the chain records which trigger was active and never a link between two. Asking for the chain overflows the stack: the run ends with `Error (R4) : Stack Overflow`. No test calls it.
- Five of the Listexuter's seven classic tests fail their own expected output, and two promise nothing. The error log of that folder records, in the sixth file, `Error (C8) : Parentheses ')' is missing`.
- A nested list that matches as a whole is reported twice.
- The article's repeating timer takes three arguments and the library takes two; the form with units is `RunEveryXT`.
- Two of the deep-lists article's path verbs, `ItemAtPath` and `ItemsAtPaths`, live on a subclass, `stzDeepList`, and not on a plain list; `Paths()` is on a plain list.
- The section-merging article presents a comprehensive verb as its point, and that verb does not exist. The two that do exist, merging what overlaps and merging what is included, run. Section merging has no test file of its own.
- There is no native engine module for the Container, Stream and Rfunction model: it runs in the library, and the engine's own reactive module is a small table of substring rules. The timeline's guard checks no gap, no overlap and no blocked span; in the classic timeline files, seven fail their expected output and eighteen promise nothing. The grid's engine-backed navigator, `stzGridNav`, has no test.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Bind a pattern to code, and process; or declare a stream and say in words what happens to the data that flows through it.
The analyst	Time is a workspace: gaps, overlaps and blocked regions are computed, and a point that would fall in a blocked region is refused with the reason.
The designer	The state of a reaction that changes a value keeps its why: the match, the computed value, the position, the trigger.
The decision-maker	Overflow is a number you can read: kept, dropped. A strategy that lost data and said nothing was found by an audit, and fixed.
The educator	The ladder is a curriculum: a pattern, a reaction, a state, a goal.
The agent	Your mind can be skills that chain on what they perceive, until the world stops changing.

9. EXERCISE

Write a Regexuter with two triggers, one for numbers and one for numbers of three digits or more, with code that turns a number into its triple. Before you run it, write under each question what you expect: what the matches are, what the results are, and whether the second trigger fires on the first one's result. Run it. If the second trigger fires, you have found something I did not.

Then say in one sentence what would have to change in the machine for it to fire, and which rung of the ladder that change belongs to.

I left this one in your hands. Take the second block of section 3 and write its two `Trigger` lines in the other order, the three-digit one first:

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() )
}
```

✎ not run: an exercise, written for you to run

Before you run it, write down whether the order changes what `Results()` and `Triggers()` hold. I do not print the output of this version, and I do not tell you the answer: section 3 says what each trigger is matched against, and section 4 says what I refused to claim. When you can say why, you will know what separates the second rung from the third.

Chapter 10 turns from the machine that thinks to the code that changes. Its title is *Programming becomes refinement*, and it holds to one principle: stop trying to govern the intelligence, and govern the change.

> **WHERE THE EVIDENCE LIVES**
>
> | | |
> |---|---|
> | 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 |
> | ON THE SITE | softanzuter |
> | ON THE SITE | craft |
> | RULINGS OF THE STUDY | 12.4, 12.18, 12.19 |

PART II · CHAPTER 10

Programming becomes refinement

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I say what I mean by refining rather than programming. It is a change to code declared to be changeable, proposed in a typed form, judged by a gate, recorded as a step, and reversible. I run the library's form of it: the refinement point, the proposal, the gate, the cascade, the trust floor and the reversal. And I print the gate as it is, five checks and not the four of my design. I say where the field leads and where the gate stops. The principle is the sentence I wrote on the back cover of my earlier book, *Refinement-Oriented Programming*: stop trying to govern the intelligence, and govern the change.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field has a name for changing infrastructure with a rehearsal: plan, then apply. A tool computes what would change, shows it, and applies it only when you say so. Terraform made the pattern familiar, and the field's admission controllers apply the same idea to a request before a cluster accepts it. More recently, the field has a name for a way of working with code-writing AI agents: specifications first, and the agent fills in. All of this is general knowledge, not a study of the record.

What these share is where they stand. The plan and the admission control are about infrastructure and requests. For the code itself, much of the field's answer to a change is still a diff, a patch of text, and a person's review of it. That worked

while every change had a person behind it who could be asked. It is a weak instrument when most changes arrive from agents, in volume; I hold this as a view, not a result.

Against that weak instrument I set one sentence.

2. THE FIRST PRINCIPLE

I wrote the sentence on the back cover of my earlier book, *Refinement-Oriented Programming*, and I have not found a better one: **stop trying to govern the intelligence, and govern the change.** The library's header says the same thing in the form of a type: a change is a typed proposal, not a diff.

The rest of the chapter runs that sentence.

3. THE SOFTANZA FORM

The practice has changed names in my life. I used to say I programmed. I now say that, more and more, I refine, and I mean something exact by it. A refinement is a change to code that was declared to be changeable, proposed in a typed form, judged by a gate, recorded as a step, and reversible. My earlier book, *Refinement-Oriented Programming*, tells the discipline through the story of Softibank, a bank facing a regulator's letter. This chapter tells what the library carries of it.

It begins by declaring where the code may change. A refinement point is a named knob written in the source as a tag, with its kind, its value and its bounds, and the code that holds it is questioned in its scene, the form of chapter 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
}
```

✓ ran: printed what this block promises

A change is then a proposal. It names a point and a value; if it clears the gate it takes effect and is recorded as one step:

HARO

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

✓ ran: printed what this block promises

The admission prints as `1` and `0`, not `TRUE` and `FALSE`: that is how this verb answers, and I print the run. And what does a proposal that does not clear the gate change, and what does the gate say? Run it:

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
}
```

✓ ran: printed what this block promises

Three refusals, and the timeline did not grow by a step! The library's comment calls this a law of its own: a rejected proposal never ends changed. Each refusal names its stage first, `constraint` twice and `structural` once, as in `structural: no refinement point 'ghost'`, so that a person, or an agent, can correct the proposal without reading a diff.

The cascade is what you read instead of the diff. Before the change, ask what it reaches. There are two cascades, and they are separate. For a point, it is the line it sits on and the other points on that same line:

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
}
```

✓ ran: printed what this block promises

`vat` sits on line 1 and also touches `[ "margin" ]`; `floor` sits on line 2, alone.

For a class, the cascade is a property of the structure of the code: who descends from it and who inherits a method. The library computes it from the code graph, and it can scan a text without touching a file:

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" ]
}
```

✓ ran: printed what this block promises

Change the rate of the tariff and three classes are in the blast radius. Two of them inherit the method, and `ImpactOf` names them: `[ "Invoice", "Receipt" ]`. The third, `Quote`, redefines it, so it owns the method and does not merely inherit it. That is the difference `ImpactOf` draws.

How does a change reach its dependents? By a rule that spans several points, judged on the state the change would produce. If the rule fails, the value is put back:

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
}
```

✓ ran: printed what this block promises

The rule refused the change and named itself: `derivation: rule 'vat-above-floor' violated -- vat must not drop below floor`. The value is still `0.30`.

And who proposed it? Every proposal carries its origin, and a point can set a floor of trust that refuses a lower origin, without refusing a person. What happens to a proposal from a language model when the floor of the point asks for a trusted origin? Run it:

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
}
```

✓ ran: printed what this block promises

The model's proposal was refused: `trust: posture :llm below the required floor for 'vat'`. The point kept `0.30`! The same proposal from a trusted origin was admitted, and the step records `trusted`.

And who may turn which knob? That is declared, too: a permission, and an authority level against a risk tier. Take a delegated actor and a knob of the third tier. Run it:

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
}
```

✓ ran: printed what this block promises

The delegated actor was refused, `'release-bot' holds 'delegated' authority (level 2)`, and the value stayed `0.20`. The same proposal was admitted once the actor was escalated! The decision is recorded: `NumberOfDecisions()` printed `1`.

Last, can a step be taken back? Reversibility is part of the data model: every admitted step keeps its prior value. Rewind to a named checkpoint, rewind to zero, go forward again:

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
}
```

✓ ran: printed what this block promises

At zero, `CanRevert()` printed `0`. One `Redo()`, and the value is `0.21`, the first step again.

That is the gate in the form the library has it. Five checks, in the order the code runs them. The structure of the proposal, then the constraints of the point, then the governance, then the trust of the origin, and last the derivation across points. I ran them above in the order of the story, not of the code. The header of the file calls this a four-stage gate in one line and five in another. The code numbers its checks 1, 2, 3, 4 and 4b. The trust check was added to the four on 16 July 2026, and the design's count was not changed. I report what runs: five checks.

Behind the gate stand decisions, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused to govern the intelligence. Much of the field's approach to AI agents is to make the model safer: alignment, guardrails, a prompt that asks for confirmation; that too is general knowledge. I refused that frame. A model's output is whatever it is, from whichever model, and the intelligence of the next generation will not hold still for me to govern it. The change is the thing that stays, and it is the thing that reaches the world. So the gate stands where the change arrives, and it asks one thing: what is proposed, from where, and does the code say this point may change that way.

I refused the diff as the unit. A diff is a text change. A refinement is a typed proposal about a declared point, with a kind, bounds and an origin. I kept the diff as an artefact, and I refused it as the thing to review.

I made the library carry the verb, and left the environment elsewhere. The refinement as a library verb, the point, the gate, the cascade and the reversal, belongs to the open foundation. The environment in which a team lives with it day to day, the screens, the live mode, the book, is a product. It does not leak into the library.

I printed the gate as it is. I wrote the design with four stages, and the code has five checks. A page that quotes my design and not my code would be wrong, and it is the code that runs.

5. THE COURT

`refinablecode_narrated` proves the point declared, the proposal that passes, the two refusals that leave the source whole (bounds and options), the cascade, the reversal, and the saved file. `refine_deepening_narrated` proves the timeline

with its checkpoints and its redo, and the trust floor. `gate_deepening_narrated` proves the refusal of an unknown point, the derivation across points with its rollback, and the governed refusal with its lineage.

Next comes the field, and where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in infrastructure, where plan and apply are the daily practice of many teams and the apply talks to real providers; this is general knowledge, as in section 1. Softanza's gate is about the change to code, which the field handles with a review.

The field's tools lead in breadth: they are not tied to one library, one language, or one declared knob. Softanza's gate judges what the code declared as a point. A change to a place that was never declared is not a refinement here; it is outside the gate.

And on trust, much of the field's approach is built to scale to any AI model. Mine rests on one declaration per point.

Now, where the gate stops.

7. THE LIMIT

BUILT	the refinement point declared in the source; the gate with five checks; the cascade of a point and of a class; reversal by step, by checkpoint and to zero; the trust floor and governed refining
DESIGNED	the experience the article describes: asking a point to explain itself, applying a plan, a one-line refinement; the cascade of the code graph joined to the gate; the refinement seat of Haro
ABSENT	a saved history: the file keeps the source and nothing of the timeline

What is built: the refinement point, the gate with its five checks, the cascade of a point and of a class, the reversal, the trust floor, and governed refining. The guards of section 5 cover them, with two exceptions. The cascade of a class has no guard here, because the code-graph guard scans a live path of the library's working tree. And no guard asserts the other points on a line.

What is designed, and absent from the library by name: the experience the original article describes. The class that article names, `stzPolyCode`, is not in the library at this commit. The one-line refinement of the design document, a point that explains itself, a plan that is applied, are not in the library. The cascade of the code graph and the cascade of the gate are not joined. And Haro has a gate in a spike written only in the former host language, with its own seat for refinement not built.

What is built, and where it stops: the limits of the gate as it stands.

- It checks the constraints of three kinds of point, a parameter, an algorithm and a library. A point of another kind, a block of code for instance, parses and passes without a check; checks for those kinds are the next step.
- An unknown trust posture ranks as trusted: the code gives it the top rank. So a proposal from an origin the code does not know clears a floor of trusted.

- A derivation is applied tentatively and restored on failure. So I say "never ends changed", not "never touched".
- The saved file holds the rendered source; the history, the postures, the timeline and the load-back are absent, and are the next step. Reversibility lives in the running object, and carrying it into the file is still to come. The library writes this file with the suffix `.zrfn`, as shipped. The Haro charter, ratified on 1 October 2026, takes the suffix `.rfn` for the refinement sentence of Haro, and leaves `.zrfn` to the library's lift from comment tags, the form this chapter ran.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Declare the places that may change, and propose changes to them as typed proposals. A refusal names its stage.
The analyst	The rule that spans several points is yours to declare, and the gate judges it on the state the change would produce.
The designer	The cascade is what you read instead of the diff: for a point, which line and which points on it; for a class, which classes inherit. They are two cascades, not yet one.
The decision-maker	Who may turn which knob is a permission and an authority against a risk tier, and the decision is recorded with its authority at the time.
The educator	Step back one, rewind to a named checkpoint, rewind to zero, go forward again. A learner can try without fear because every step keeps its prior value.
The agent	You may propose. Your origin is on the proposal, and a point can set a floor your origin does not clear.

9. EXERCISE

Take two settings of a program of yours: a rate and a ceiling, say, with a rule that the rate must stay below the ceiling. Declare them as points, with bounds, and declare the rule across them. Before you run anything, write under each proposal whether it is admitted and, if not, at which stage it is refused. Propose a change of the rate above the ceiling, and one from an origin you declare as an AI model, against a floor you set. Then rewind to zero and go forward one step.

Run it. If you were wrong about the stage, you have found a place where your own gate disagrees with your own expectation.

I left this one in your hands. Section 7 says what the gate does with an origin the code does not know, and the trust block of section 3 is the block to run. Its first lines are below, and `d` is the object of that block, so run section 3 first; change one thing, and replace `:llm` with a name of your own, one the code was never told about.

HARO

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

✎ not run: an exercise, written for you to run

Run it, and read what the gate does against what section 7 says it does. I will not tell you whether the gate is right. When you have decided, you will know what the trust floor does with an origin it has never met.

The next chapter states a law: a language is declared and closed, and whatever speaks it, a person or a model, is judged by it. Here the thing judged was a change; there it is whoever speaks.

> **WHERE THE EVIDENCE LIVES**
>
> | | |
> |---|---|
> | NARRATION | `stzpolycode-refinement-centered_programming.md` |
> | ON THE SITE | refinement |
> | RULINGS OF THE STUDY | 12.19, 12.23, 12.29 |

PART III

Languages, rethought

> *A language is declared and closed, and whatever speaks it, person or model, is judged by it.*

11 · Law: a language is declared and closed, and whatever speaks it, person or model, is judged by it
12 · A language, declared
13 · Natural, and executable: two directions, one discipline
14 · Knowledge programming
15 · Pattern languages
16 · Seven languages, one door

PART III · CHAPTER 11 · SPINE

Law: a language is declared and closed, and whatever speaks it, person or model, is judged by it

DRAFT

Run on 2026-10-15 12:58 UTC+01:00 inside the library at commit 93d7a39bd: 11 of 11 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I state the law of Part III and run it. I begin with a sentence that runs because every word of it is in a dictionary the library owns, in English, Hausa, French and Arabic. I show what the library says about a word it does not know. Then I declare the shape of an answer as a schema, judge answers against it, and show the one thing a grammar cannot kill: a false answer in the right shape. I say which guards prove this, where the field is ahead, and where the natural layer is not yet closed, which the law states. The law itself is one sentence: a language is declared and closed, and whatever speaks it, a person or a model, is judged by it.

WHAT THE AI AGE CHANGES HERE

In Part II, the age changed who writes the code. Here it changes who speaks the sentence. A sentence of a language used to have a person behind it, someone who could be asked what they meant. Today the writer of a sentence can be an AI model, which cannot be asked, and which will write a fluent sentence whether or not it is true.

So the grammar must be closed, and the court, the part that accepts or refuses a sentence by its grammar, must read it. A language whose sentences may contain anything cannot refuse anything, and a court cannot judge a sentence it cannot

read. The vision of the platform puts the whole movement in one line, which I ratified: humans and machines meet in declared languages, small, closed and judged, not in general-purpose code.

This is also where the order of the book turns around. The cover goes from the familiar to the new: programming, languages, intelligence. The record's logic goes the other way. The first act is to declare a language. Programming, the discipline of Part II, happens inside declared languages: the forms of the vocabulary are already a declaration in Haro, in a spike, a short experiment written to try a design, outside the library. Intelligence, Part IV, is governed by both: a model is held to a declared grammar before it speaks, and its sentence is judged after. The book keeps the cover's order, so this chapter sits in the middle; in the record's order it would come first.

1. THE CONCEPT, AS THE FIELD HOLDS IT

For a working programmer, a language is a general host, Python, JavaScript, Java. A "domain language" is usually a library whose calls read like sentences, while the file around them may contain any code of the host. The estate's own study of the field, written in August 2026, names what happened to those designs. Internal languages died of openness, because a build file is declarative only until someone writes arbitrary code in it. Language workbenches died of the weight of their tools. And it names the languages that won: SQL, regular expressions, Make, HTML and CSS, the configuration languages. Its verdict is that they won because they are closed.

The model era added one practice: when an AI model writes, either check what it wrote, or constrain its output while it writes, with a grammar or a schema. The reference implementation of grammar-constrained sampling is an open project, and the record says so where it uses the same algorithm. And in the field's common language today, "programming in natural language" means one thing: prompting a model.

2. THE FIRST PRINCIPLE

A language is declared and closed, and whatever speaks it, a person or a model, is judged by it.

3. THE SOFTANZA FORM

Can a sentence run without an AI model guessing what it means? It can, when every word of it is in a dictionary the library owns:

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 │
#    ╰─────────────────────╯
```

✓ ran: printed what this block promises

Four lines of English became a rounded box around spaced capitals.

The dictionary is data, keyed by meanings, not by English words. So can the same meaning be said in another language by adding a block of data, not code? Here is one instruction in English, in Hausa, in French and in Arabic. The Hausa sentence is the narration's own, and the French one comes from the library's guard with its data changed. I composed the Arabic one from the library's vocabulary. No native speaker has read the Hausa or the Arabic as far as the record shows:

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
```

✓ ran: printed what this block promises

Four languages, one meaning, and `HELLO` four times.

And a program says back what it understood, in its own language. A person reading Arabic can check the machine's reading of their sentence without reading a line of 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
```

✓ ran: printed what this block promises

The program said back `أنشئ قائمة بـــ [ 5, 3, 5 ] -> أزل التكرارات`: the list it was given, and the removal of duplicates.

Now the part the library does by rule: say what it did not understand. What does it do with a verb it does not know? Misspell one and run it:

HARO

```
o1 = Naturally("Create a string with 'softanza' Uppercse it")
? o1.Result()
#--> softanza
? @@( o1.Unresolved() )
#--> [ [ "Uppercse", "uppercase" ] ]
? o1.UnderstoodAll()
#--> 0
```

✓ ran: printed what this block promises

The misspelled verb is not guessed. `Unresolved()` answers `[ [ "Uppercse", "uppercase" ] ]`, the word it met beside the word the library thinks you meant, and `UnderstoodAll()` answers `0`: the program admits that it did not understand everything. That is the permissive mode: an unknown word is skipped and reported. An AI agent gets the strict mode, which refuses, and a world, a declared list of allowed operations, can limit its sentences to that list:

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
```

✓ ran: printed what this block promises

Strict mode refuses `'Fooify'` and still names the nearest word: `did you mean 'boxify'?`. In that world, a sentence that reverses a list is refused! The refusal names the operation: `Operation not permitted in this world: METHOD_REVERSE`.

The other direction of natural code in Softanza is the near-natural chain, where the person is the parser and the chain reads as a sentence. When it answers no, does it say why?

HARO

```
? IsQ().TheQ().LengthQ().OfQ("softanza").LessThanQ().TheQ().LengthQ().Of("Haro")
#--> 0
? Why()
#--> no: 8 is not less than 4
```

✓ ran: printed what this block promises

The chain answered `0`, and `Why()` gave the reason: `no: 8 is not less than 4`. A verdict says why.

Now the law at its sharpest: an AI model. The library declares the shape of an answer as a schema. What happens to a schema that names a type the library does not have? It is judged when it is declared, before any model is called:

HARO

```
try
        StzOutputSchemaQ([ [ :field = "x", :type = :colour ] ])
catch
        ? cCatchError
done
#--> stzOutputSchema: unknown field type 'colour' (use
string|number|boolean|oneof|list|structure).
```

✓ ran: printed what this block promises

The refusal names the unknown type, `'colour'`, and lists the six it accepts: `(use string|number|boolean|oneof|list|structure)`. The schema was refused before any model was called! A correct schema is a small closed language. The court judges any answer against it, whoever wrote the answer. And the schema emits its grammar, so that a model can be held to it while it writes:

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")
}
```

✓ ran: printed what this block promises

Look at the four verdicts. The first answer printed `1`. The second, with `mood: angry`, and the third, with `age: 900000`, printed `0`: the court refused both. The fourth printed `1` again. Hadiza is forty-one in the first answer and sixty-two in the fourth, and the court accepts both! A grammar kills malformed sentences, not false ones. The record says it in five words: structure kills malformedness, not falsehood. This book says the same: the truth of an answer is not for the grammar. The record says that truth needs reversibility, an auditor and a Principal, the person who rules. A grammar is the gate, not the judge of everything. The grammar also prints a `boolean` rule that no field of this schema uses.

A template is a contract too. A sentence with an unfilled hole is refused before it runs:

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.
```

✓ ran: printed what this block promises

And patterns are small languages of their own, though not yet closed ones: chapter 15 shows the pattern classes accepting a word they do not know, without complaint. The shape of a list is declared, not written as a loop:

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
```

✓ ran: printed what this block promises

Two shapes matched, two did not, and no loop was written.

The chapters of this part prove the law from five sides. Chapter 12 tells a language declared: the forge (the machinery that produces a language's parser, court, printer and grammar from its declaration alone), the closed grammar with no escape into the host, and the court born with the grammar. It tells too the grammar emitted so that a model can only speak valid sentences, and the parts a declaration yields. Chapter 13 tells the two directions of natural code, the machine as the parser and the person as the parser, in one discipline. Chapter 14 tells knowledge programming, one file that is a system. Chapter 15 tells the pattern languages, regex lifted to lists, numbers, tables, matrices, time and graphs. Chapter 16 tells the seven languages Softanza reaches through one governed door, and why a door is not a facet.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I chose the act: declare a language. Not "use our language". Declare yours: the language of your domain, in your words. That is the kernel of the platform, and it is the reason this part exists.

I refused the open escape. The meta-language that declares languages in the estate lives outside this library. Its founding rule refuses to define any form that would let a sentence reach host code, call a function, evaluate a string or touch a file. Not forbidden: unexpressible. An internal language that cannot refuse arbitrary host code is, in the words of the estate's field study, an API wearing a language's costume.

I refused natural language as a way to declare. Natural language asks; it never declares. A model may draft a declaration, and by the vision's rule, which is designed and not yet built, the draft enters the world only as a proposal: the model proposes, the court decides, a person ratifies.

I kept the promise of 2020 and changed its mechanism. My first slide promised executable natural language. The vision of 2026 judged that slogan: it promised more than a court can defend, because a natural sentence can be ambiguous and a lie can be grammatical. So the naturalness stays at the door, where a sentence asks or drafts, and the execution belongs to closed languages a court can judge. I built the natural layer that way long before the vision named it. Only words defined in the dictionary are interpreted. Every other word is skipped and reported, or refused in strict mode, never guessed.

I took governance out of the library and kept the refusals in it. In July 2026, a kit for governing AI agents landed inside the natural layer. I removed it the same day: the strict mode, the refusals, the why and the understood belong to the library; a product for governing agents does not. And I removed a set of markers, suffixes that marked a reference, which chapter 13 tells: their letter-order

cipher had been inverted by its own inventor, from memory, and they duplicated meanings. A language a person speaks never marks a reference that way. An ungrammatical chain that runs is a bug.

I refused a second embedding. Chapter 1 states that decision; here I add only that a door brings a foreign runtime, a facet brings a familiar syntax over the same grammar, and chapter 16 tells the difference.

5. THE COURT

`multilingual_pack_narrated` proves the languages as data: the packs, the programs in French and Arabic, a pack added at runtime, and the program paraphrasing back in its own language. `semantic_resolver_narrated` proves the unknown word with its suggestion, the strict mode, and the allow-list under which everything off the list is grammatically impossible. Its run does not yet reach its end at this commit. I ran it twice on 8 October 2026, for chapter 13, and each time it stopped with an error raised from a string-splitting function, before its last assertion. So the page counts its assertions and does not claim its verdict. `literate_templates_narrated` proves that a template is a contract. `question_narrated` proves that a yes-or-no verdict records its why. `structuredoutput_narrated` proves the court of structured output, including the lie it lets through. `schema_grammar_narrated` proves the grammar compiler, and `constrained_decoding_narrated` the sampler that holds a model to it.

Seven guards prove the law; the next section says where the field is ahead of it.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in tools. The language workbenches remain ahead on the axes the estate's design deliberately gave up: projectional tooling (editors that understand a language as you type), IDE integration, macro-theoretic depth. The estate's study says it in its own words: that is a bet, and it is named as one.

The field built constrained decoding first. Softanza's sampler follows the reference implementation's algorithm and vendors nothing. Measured against one small model of a hundred and thirty-five million parameters, on ten prompts, the answers were valid at the first attempt two times in ten when checked afterwards. Held to the grammar, the model was valid ten times in ten. That is one small model and ten prompts, and nothing in the record measures a large one.

And the age cuts both ways. Generated code makes careless language-making cheap too. That is why the court is not an accessory of a declared language. It is the core. This is my view.

7. THE LIMIT

BUILT	the natural layer's vocabulary as data in five languages, with its refusals; the court of structured output, its grammar compiler and its sampler; the pattern languages, which accept a word they do not know
DESIGNED	the forge that produces a language's parser, court and grammar from its declaration alone; a grammar emitted for every declared language; the writing style as a declared grammar
ABSENT	a natural layer closed all the way down: one form of list still reaches host code; pattern classes that refuse a word they do not know; the standard Hausa letter ɗ in the Hausa pack

What is built: the natural layer, its vocabulary as data in five languages, English, Hausa, French, Arabic and Turkish, and its refusals. Built too: the court of structured output with its grammar compiler and its sampler, and the pattern languages. These accept a word they do not know today; closing them is to come.

What is designed: the forge that would produce, from a language's declaration alone, its parser, its court, its printer (the part that writes a sentence back out) and its grammar for a model. The forge is built outside the library and designed for the library: a first sketch of it was built outside the library in October 2026. The library cannot run it yet. The vision's text says the forge has not yet forged (made a language's parts from its declaration). The sketch has since forged courts and

grammars for several languages, judged by those languages' own fixtures, and the vision has not caught up. Chapter 12 tells it with its limits. A grammar for every declared language is designed and to come.

What is absent, and the law states it. The natural layer is not closed all the way down. One form, a list written inside brackets, is still handed to the host to evaluate. Does a host function inside the brackets run? Try one:

HARO

```
? @@( Naturally("Create a list with [ upper('x'), 1 ] and remove its duplicates").Result() )
#--> [ "X", 1 ]
```

✓ ran: printed what this block promises

The function `upper` is not a word of the natural dictionary, and the list came back as `[ "X", 1 ]`. It ran anyway! The record already knows this escape and has routed its repair. Until the repair lands, "the natural layer is a closed language" is true of its words and not of its lists, and this book will say so.

And the Hausa pack does not yet read the letter ɗ, which standard Hausa spelling uses. The pack stores it in an encoding that does not carry it yet. When I probed it on 8 October 2026 on the snapshot before this commit, a sentence written with the letter was not read, and the pack did not report it. The Hausa in this chapter is written without the hook, the way the pack reads it today. The record holds no reading of the pack's sentences by a Hausa speaker. The foreign-language sentences printed in this chapter come from the library's narration and guards, or are composed from its vocabulary. No native speaker has read them as far as the record shows.

These are the limits; the next section opens six doors on them.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You keep writing code. What you declare is a closed grammar, and the court judges your sentence and an AI agent's by the same rules.
The analyst	Your domain's words become a data pack, and the system understands them or refuses them with the reason.
The designer	The refusals are part of the language: what is off the list is refused in strict mode, so it never needs a warning.
The decision-maker	An AI model held to a declared grammar answers in the right shape. It can still answer a falsehood in the right shape, and the record prints that limit with the result.
The educator	A sentence in Hausa, French or Arabic runs and says back what it understood, so a learner reads the machine's reading of their own words.
The agent	You may speak only the declared language. Strict mode is your posture, and your draft is, by the vision's rule (designed), a proposal the court judges.

9. EXERCISE

Declare, with `StzOutputSchemaQ`, a record of three fields from a domain you know: a loan request, for instance, with an amount inside a band, a currency from a closed list, and a purpose in free text. Before you run anything, write three answers an AI model might give: one the court must accept, one it must refuse because of its shape, and one it will accept although it is false. Under each, write your prediction as a promise.

Run it. The third answer is the lesson of this chapter. Write down, in one sentence, who in your domain must judge it, since the grammar cannot.

I left this one in your hands. Section 7 says that a list written inside brackets is still handed to the host, and section 4 says why a language must refuse any form that reaches host code. Run the section 7 block with one change, a different host function inside the brackets:

HARO

```
? @@( Naturally("Create a list with [ lower('X'), 1 ] and remove its duplicates").Result() )
```

✎ not run: an exercise, written for you to run

Look at what the library does with the function, and say in one sentence what a closed natural layer would have done instead. When you have, you will know why "the natural layer is a closed language" is true of its words and not yet of its lists.

Chapter 12 is *A language, declared*. Its principle: declaring a language is as easy as declaring a variable.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	languages
RULINGS OF THE STUDY	12.4, 12.10, 12.16, 12.19, 12.21, 12.27, 12.28, 12.29, 12.30

PART III · CHAPTER 12

A language, declared

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 8 of 8 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take one declaration of a restaurant's order and give it to three readers: a person, an AI model and the court. I let the same declaration produce a grammar, and I show where the grammar and the court agree and where they part. Then I leave the library for the forge. It is the machinery that turns a language's declaration alone into its court and its grammar, and I say what it has forged and what it has not. I decided that Haro is the language of languages, and I refused an open escape form. The principle is one rule of Haro: declaring a language is as easy as declaring a variable.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Two families of tools let you declare a language in the field. The first is the language workbench and the parser generator: JetBrains MPS, Xtext, Spoofax, and Racket's `#lang` give you the means to define a language with its grammar, its editor and its checks. The estate's own field study of August 2026 says why they did not win. They depended on an ecosystem, and Racket's languages live and die inside Racket.

The second is the internal DSL: a library whose calls read like sentences, while the file can still hold any host code. The same study has the sentence I keep: closedness is the product, and an internal DSL that cannot refuse arbitrary host

code is an API wearing a language's costume. Its control group is the closed languages that won: SQL, regex, Make, HTML and CSS, HCL, GraphQL, Markdown, CUE.

For the grammar a model must obey, the record names one tool: the grammar sampler of a well-known local inference library, which the library's machine follows for its element encoding and its algorithm. Nothing of it is vendored. It is general knowledge, not a study of the record, that model vendors offer structured-output modes that take a JSON schema, and that open libraries turn a schema or a regular expression into a mask on the model's tokens.

Chapter 11 told what a closed language is, and what was lost when the field left it open. I take that as read. This chapter says what is new: what *produces* a closed language.

2. THE FIRST PRINCIPLE

Declaring a language is as easy as declaring a variable.

It is the heading of one rule of Haro, ruled on 1 October 2026, and the kernel of the vision says the same: the era makes declaring a language as easy as declaring a variable. The other half of the chapter has its sentence, from the alphabet's totality doctrine, ratified on 27 September: malformedness dies at the sampler, falsehood dies at the court, and non-termination is unexpressible.

And the one I must carry beside every win, from the library's contract for structured output: structure kills malformedness, and not falsehood.

3. THE SOFTANZA FORM

Take a declaration of what a restaurant's order must look like. It has three fields: a dish from a closed menu, a quantity between 1 and 12, and an optional note. One text is read by three readers. A person reads a description. A model reads a clause for its prompt. And the court reads the same text and judges an answer. What does it answer to a dish called pizza and forty portions? Run it:

```
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'.
}
```

✓ ran: printed what this block promises

The court answers in the declaration's words: `field 'dish' is a closed enumeration (margherita | tiramisu) and the model answered 'pizza'.` Now the same declaration produces a grammar. The court and the grammar are born of one text, and they agree on the shape. They part ways on a value. Forty portions has the right shape: which of the two lets it through, and does the library say what the grammar dropped? Run it:

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
```

✓ ran: printed what this block promises

The second pair is the point of the chapter: the grammar answers `1` and the court answers `0`! I do not print the list of what the grammar dropped: its text carries, in one place, the court's former name, and that name belongs to the history in Appendix H. I test that the list names the limit of the range.

An AI model is not asked to be careful. The sampler asks the grammar at every candidate token whether it could continue. So a word that is not on the menu is not refused after it is said. Can it begin at all?

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
```

✓ ran: printed what this block promises

It cannot: `dish: p` answers `0`. The last line is a failure the measurement of section 5 found in a small model: the placeholder `<number>` echoed where a number belongs. And a token is not a byte. A piece may be valid as a prefix and invalid as a completion, and the machine is fed the whole piece. What does it answer to `yesterday`, which begins like `yes`?

HARO

```
cY = 'root ::= "yes" "\n"'
? StzGrammarCouldContinue(cY, "y")
#--> 1
? StzGrammarCouldContinue(cY, "ye")
#--> 1
? StzGrammarCouldContinue(cY, "yesterday")
#--> 0
```

✓ ran: printed what this block promises

The pieces `y` and `ye` answer `1`, and `yesterday` answers `0`!

What the machine will not hold, it refuses by name. A closed language refuses what it cannot carry, and never approximates. A left-recursive rule is refused by a code, and so is a rule referred to and never defined. A nested structure is refused by the grammar. Does the court still read it?

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
}
```

✓ ran: printed what this block promises

The text of the nested-structure refusal carries the court's former name too, so I print only the codes. The last line is the point: the court reads what the grammar refuses to write.

The declaration language is itself closed. A typo in the law is refused when the law is declared. The library's contract says why. A key such as `:requird = 0` would leave a field required without a word, and that is the class of typo a court must not tolerate in its own law. What does the court answer to one?

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).
```

✓ ran: printed what this block promises

The typo was refused at once, and the message lists the keys that are allowed!

And a host function is not a word of the closed language of conditions. There are two ways to say a condition, the closed language and the door of a function. A square root is a word of the engine's expression language, and a sine is not. What does the textual form answer to the sine, and what does the function form, which runs host code?

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 ]
}
```

✓ ran: printed what this block promises

What I saw, and the narration does not say: the refusal is silent and whole. The textual form printed `[ ]` and the function form printed `[ 1, 2, 3, 4, 5 ]`: nothing from the one, everything from the other. A condition with the true left branch and a sine on the right printed `[ ]`. A condition that calls a function of mine never called it. A syntax error also printed `[ ]`. The language is closed, and it refuses without a word. Chapter 11's natural layer says why it refuses, and this one does not.

THE FORGE, OUTSIDE THE LIBRARY

A language declared once should give its court, its mouth, its voice and its readers. That is the forge, and it is where the study of this book meets its own limit: the forge is not in the library.

I will say what the record holds, and what I observed on 8 October 2026 on the older runtime, the oracle. The record is a charter and a spike in another repository, in pure core code that runs on the oracle. The forge reads a declaration and writes two files, a court program and a grammar. The spike's rungs, between 1 and 4 October 2026, forged courts and grammars for the machine language of the estate and for a fleet language declared over it. They did the same for the expression language and for four more letters of the alphabet, the vision's family of declared languages, each known by a letter. Each was judged against the pre-existing language's own cases. Those cases were unadapted for the expression language (W) and for Z, re-expressed for F and A, and, for Q, the older language's own answers, captured. In order: 117 of 120 for the machine language, the three left being the fleet file's kinds. Then 32 of 32 for the fleet language, 29 of 29 for W, 16 of 16 for Z, 25 of 25 for Q, 26 of 26 for F and 35 of 35 for A. This is chapter 3's method, declare then govern, applied to the forge itself.

I ran the forge on a small language I wrote for this book, a table's order in the words of the dining room. It is an example, written for the book and run once, outside the library:

```
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"
```

The forge printed that it had forged one kind and one refusal, in a fraction of a second of wall time, and wrote a court of about 2.6 kilobytes. What did that court answer to a dish off the menu, to forty portions and to an alien verb? It refused all three, in the declaration's own words:

```
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.
```

The refusals are the declaration's own words, written beside the grammar. Even `DROP TABLE orders` was refused in the dining room's words! This is the whole of the evidence for the `order` language: it ran once, outside the library, and it is not a declaration of the record.

Three more observations, each a limit.

The meta-court refuses the escape. The declaration of the language of declarations lists the operators and operands it may use. A declaration that listed `eval` among its operators, in a fixture pinned on 29 August 2026, was refused:

`eval` is not in the closed operator menu, an escape hatch is unexpressible, not forbidden. A host call among the operands was refused in the same way.

The forge's grammar holds shape only. A menu seat comes out as a general identifier, so the forged grammar lets `pizza` through. The library's own schema compiler, in the blocks above, closes the same menu into an alternation. On the estate's machine language, the forged grammar accepts a dangling reference that the court refuses. The record names a difference between mouth and court, the part that a context-free grammar cannot say: duplicate names, dangling references. I found no record that states the menu gap as a gap.

The vision has not caught up. The vision's text, dated between 27 September and 5 October 2026, says the forge has not forged. The principles file, dated 1 October 2026, says that the forge does not exist. The same repository built the spike the same day and forged for several languages. So the true sentence has two sides. The forge has forged courts and grammars for several languages, outside the library, and it has not forged the full list the principle promises. And the vision's text has not been updated. In the book's stage words, the forge is built outside the library and designed for the library, and chapter 11's limit says the same.

THE PARTS OF A DECLARED LANGUAGE

The charter lists what the forge produces from a declaration, its parts, and I give each a stage word:

PART	WHAT IT IS	STAGE
the text	the declaration, plain and closed, with a rationale required on every declaration	built outside; the library's analogue is the schema declaration
the court	the judge generated for the language, speaking in the declaration's own words	built outside; the library's analogue is `Verify` and `Accepts`
the mouth	the grammar for a constrained AI model	built in the library for schemas; built outside for declared languages, shape only
the prompt	the instruction an AI model reads, tuned by measurement	built in the library; not known for declared languages

PART	WHAT IT IS	STAGE
the person's reading	a description for a human	built in the library (`Describe`)
the voice	the language narrating a run, a sentence, a refusal; never stored and never parsed back	built outside; the library's analogue is `Understood()`
the tongues	a human language as a data file	built outside for two tongues; built in the library for five packs (chapter 11)
the projections	the world rendered to an application, a server, a page, a device	designed
the facets	a second syntax of Haro	designed; chapter 16 tells it

The word *facet* has two meanings in this book, and I define the one of this part here. A facet is a second syntax of Haro, for a programmer of another language: another spelling of the one grammar, and never a runtime. It is the last row of the table, and chapter 16 tells it. The other sense is a tab of the environment's ribbon, one phase of a life, which is chapter 26's.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I decided that Haro is the language of languages. The charter's header quotes me, from 1 October 2026: Haro is a language of languages, in which the crafters define a language for their domain and program in it, or let their AI agents. The charter records a reversal: its first decision, as the general-purpose floor under the family of declared languages, was reversed. The rest of that ratification was my "go with your recommendations" on seven decisions, and those are rulings by delegation.

I decided to declare, and not to write. The act of the kernel: declare a language. It was extraction and not invention: the first languages were declared after they already ran and had consumers.

I refused an open escape form. No clause, operator or operand denotes host code, calls, evaluation or input and output. Unexpressible, not forbidden. Two fixtures pin the refusal, and I re-ran them.

I refused logic in the declaring surface. The declaration declares and does not compute. The alphabet is deliberately not Turing-complete, and that is its power and not its poverty. This is my view.

I refused natural language as the declaring surface. Natural language asks, and it never declares. A draft may be proposed, and the court decides, and the human ratifies.

I refused one text for declaring and for computing. Programmability and closure never occupy the same surface. Computing code reaches a declared world only through the doors the forge generates, and chapter 16 is about the door.

I refused to let the grammar half retire the court. The contract says the court does not retire when constrained decoding lands. And what the grammar cannot express it refuses, and never flattens or approximates.

I refused a second facet as a second meaning. A facet spells the graph; it never gives it Python's or JavaScript's meaning. That is a session's memo, which proposes and rules nothing, and its question of how many facets before the announcement is mine, and unanswered in the record.

Next is the court, and the guards that stand behind it.

5. THE COURT

`structuredoutput_narrated` proves that a declaration is judged when declared, with an unknown type, an unknown key and an empty menu refused. It proves that two wire shapes are read and nothing is guessed, that there is no partial credit, that nesting works, and that a retry fits in one budget. In its tenth scene, it proves that a lie is let through on purpose. `schema_grammar_narrated` proves that the declaration compiles to a grammar, and that a closed enumeration becomes a closed alternation. It proves that optional and list arity are handled, that the unenforced constraints are listed by field and operator, and that a nested structure is refused with a named code. `constrained_decoding_narrated` proves that the machine walks a grammar byte by byte, that a prefix is not a whole, and that `yesterday` is refused. It proves that the placeholder the model wrote cannot be reached, that refusals are named, that structure is not falsehood, and that a grammar on a streaming session is refused. Its two scenes that need a generative model say so. `483_w_and_wf_forms_in_conditional_code` proves that the two forms of a condition give the same positions. It does not assert the closed language's refusal, and no guard does.

No guard in the library tests a forge, since there is none. The forge's guard is outside: the spike passes twenty-three stages in 367.5 seconds on 4 October 2026, by its own file. I did not run it. And the meta-court's own first run was twenty-eight of twenty-eight on 29 August 2026. The book's guard reader resolves only guards in the library's test tree, so these cannot be named in the front matter.

The first attempt is valid in two cases of ten when checked afterwards, and in ten of ten when constrained. That figure is a number in a comment of these guards. It cannot be re-taken here: the live scenes need a generative model file, and the snapshot holds only a checksum file in its folder for models. Chapter 11 carries the figure, and I point to it.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The workbenches lead on tooling. The field study says it itself: they remain superior on the axes the design deliberately abandons, projectional tooling, editor integration and depth of macros. And the bet that those axes are the wrong ones is named as a bet.

Generated code makes undisciplined language-making cheap too, and that is why the court is the core and not an accessory.

Constrained decoding: the field built it first, and the reference implementation is another library's. What the record adds is the pairing: a grammar and a court from one declaration, and the list of what the grammar dropped.

A grammar over-approximates a court. Duplicate names, dangling references and the coherence of a menu are not context-free. I observed it both ways above: a band the grammar cannot carry, and a reference it cannot carry.

The code a forge emits must be read by someone. The family style that every declared language inherits has a success test, that learning the second language should take an evening, because it is the first language with different verbs. It is a design requirement and not a measurement. No learner has been timed in the record I read.

The limit comes next, at the edge of a second closed language.

7. THE LIMIT

BUILT — in the library: a declaration judged when it is declared; a court and a grammar from the one declaration, with the list of what the grammar dropped; a grammar machine that walks bytes and refuses by name; a description for a person and a prompt for an AI model from the same text. Outside the library: a forge that turns a declaration into a court and a grammar, run on the estate's own languages

DESIGNED — the forge inside the library; its full list of products from one declaration (printer, knowledge graph, natural door, narration, runtime, one static binary); the projections of a language to an application

ABSENT — the forge's grammar for a menu word; a guard that asserts the refusal of a host function in a condition; the figure two of ten against ten of ten, which needs a model file the snapshot does not hold

The library has two closed formula languages, and they behave differently at the edge. A reader who takes the closed condition language of chapter 11 and of the condition above to cover every formula will be wrong at the table. The calculated column of a table falls back, by its own header, to the classic evaluation path when the engine cannot compile a formula. Does the word `sin` that the condition refused run here?

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 ]
```

✓ ran: printed what this block promises

The sine the condition refused, the table computed, and it printed `[ 0, 0.84, -0.76 ]`! I also saw, and do not show, a second thing. On another probe table, with A as 1, 3 and 10 and B as 2, 4 and 20, I wrote a formula `@(:A) + HostBonus(@(:B))`. The function of mine inside it multiplies by a thousand. The formula printed `[ 2001, 4003, 20010 ]`: a user-defined host function ran from inside a formula string. And a formula that names a function that does not exist raised the host's own error. So the second closed language is open at its edge. No guard asserts either behaviour.

What is built: in the library, a declaration judged when declared; a court and a grammar from it; the list of what the grammar dropped. A grammar machine that says it is constrained; a prompt and a description from the same text. The build counts 151 assertions in three narrated guards. Two closed little languages of conditions, with limits. The natural layer's tongues as data.

What is built, but outside the library: a forge from a declaration to a court and a grammar, in the Haro charter's repository.

What is designed: the forge producing from a declaration alone the parser, court, printer, grammar, knowledge graph, query grammar, natural door, narration, runtime and one static binary. By my reading of the charter, the spike has shown the court, the grammar, a runtime that walks the declaration, a query grammar, a narration and the court of the natural letter. I did not find shown a printer for declared sentences, a knowledge graph forged from a declaration, or the one static binary per language. The library having a forge: nothing in it declares or forges a language. A grammar for every declared language shipping in the library: promised in the kernel, not shipped. And the projections: the world rendered to an application, a server, a page, a device.

What is absent: the forge's grammar for a menu word, which I saw above holds shape only, and the figure of two of ten against ten of ten, re-taken. And any record that the forge runs on the Haro runtime. At the time of the vision's writing the language did not exist; the charter's runs are on the older runtime and on the oracle.

The six doors follow, from the programmer to the agent.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You declare the shape in a few lines and get its court, its grammar and its prompt from them. What you do not get is a way to run a host function from the declaration: the language has no word for it. The table's formula language of section 7 is another language, and open at its edge.
The analyst	The declaration is plain text in the domain's words, and the court answers in those words.
The designer	The refusal is declared beside the grammar, in the domain's voice, and is not authored per runtime. A designer chooses what a closed menu says when it is broken.
The decision-maker	One declaration produces the mouth and the court, and the forge itself publishes where they disagree. The forge is outside the library today. Do not buy it as shipped.
The educator	The reader can predict, byte by byte, what the sampler will allow, and learns that a grammar kills malformedness and not falsehood.
The agent	You speak only the grammar you were handed. A sentence you draft is a proposal that the court judges. Your mouth is shape, and your falsehoods are the court's and a person's.

9. EXERCISE

Write, by hand, a grammar for one line of a lunch order: `dish:` then one of two dishes of your choice, then a newline. Before you run anything, write your predictions as promises:

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) = ""
#--> ?
```

✎ not run: an exercise, written for you to run

Then add a number to the line, `quantity:` and digits, and write a sixth prediction: does your grammar accept `quantity: 4000`? Run it. The lesson is the third, the fourth and the sixth. The machine already knows that a prefix could continue and is not yet a sentence, and it cannot know that the number is too large. Write down, in one sentence, who in your restaurant must refuse forty portions, since the grammar cannot.

I left this one in your hands. Section 3 ends its run of conditions on a refusal without a word, and section 4 says why computing stays off the declaring surface. Take the two lines that put the same question to the list in the two forms, and change one thing: write another number in place of the 2, the same in both lines. I print nothing under them.

HARO

```
o = new stzList([ 4, 7, 10, 3, 8 ])
? @@( o.FindW('{ sin(@item) < 2 }') )
? @@( o.FindWF( func x { return sin(x) < 2 } ) )
```

✎ not run: an exercise, written for you to run

Run them for several numbers. Look for a number at which the two forms print the same thing, and say whether the printout alone tells you which of the two refused and which answered. When you have, you will know what it costs that a closed language refuses without a word.

The next chapter turns to the natural sentence, in two directions and under one discipline. A sentence runs because every word of it is in a dictionary the library owns, and what the library does not know, it refuses, with the reason.

WHERE THE EVIDENCE LIVES

NARRATION	stz-conditional-code-narration.md
NARRATION	stznatural-vs-ring-naturallib.md
NARRATION	stznatural-narration.md
NARRATION	stz-bridging-minds-and-code.md
ON THE SITE	languages
RULINGS OF THE STUDY	12.10, 12.21, 12.27, 12.29, 12.30

PART III · CHAPTER 13

Natural, and executable: two directions, one discipline

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 17 of 17 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I run the two directions of natural language in the library: the chain, where you are the parser, and the sentence, where the library is. I print the refusals on both sides, five language packs as data and a sixth added at runtime, and the count of what each pack can say. I say what I refused, the markers on reference and the ungrammatical chain among them, and where the dictionary's own court fails at this commit. The principle is one sentence. A sentence runs because every word of it is in a dictionary the library owns, and what the library does not know, it refuses, with the reason.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In the field's common language, "programming in natural language" means one thing: asking an AI model for code, or asking it to do the task, in a prompt. The tools are chat assistants and code completers, in wide use since about 2021. What the sentence runs on is the model's guess. The same sentence may give different code on different days, the model needs a network or a local runtime, and a person reads the result to know what was understood. Chapter 11 said it, and I point back.

Older practices exist, and the record names them. Inform 7, a language whose programs are English sentences, compiled whole, with a world model and a way to say *it* across sentences. AppleScript, English-like and hard to write, because its naturalness was on the surface and not in a semantic model. Controlled natural

languages, whose strongest device, the library's own comment says, is showing what can be said next. A session's reading of 30 August 2026 is that Inform 7 is a whole-text compiler with referential context, and not a command dispatcher. The dates are general knowledge, not a study of the record.

None of them is what the record claims. The claim is the opposite of prompting: a sentence executes because every word of it is in a closed lexicon, defined as data, mapped to a verb the library documents. Anything else is refused with the reason. And it has two directions, which the library's design review tabulates. In the first, the machine is the parser. In the second, the human is.

2. THE FIRST PRINCIPLE

A sentence runs because every word of it is in a dictionary the library owns. What the library does not know, it refuses, with the reason.

My study of the record, and the lede of the site's page of 8 October 2026, say it. For the second direction, the design review's thesis: in near-natural language, the human is the parser. And the older motto, which chapter 5 carries: what you think is what you write.

3. THE SOFTANZA FORM

I start with the second direction, because it is the older.

THE HUMAN IS THE PARSER

A near-natural chain is a sentence in which every link is a real call. The sentence is "the word haro is a lowercase Latin word with a length of 4 letters and only 2 vowels". When it is false, what does the chain say? Run it:

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
```

✓ ran: printed what this block promises

The chain answers `no: expected exactly 1, found 2`. Degree words are devices, and the reason is kept. The block reads it with the console's `Why()`; the chain also keeps its own, `WhyCheckFailed()`, which I do not call here:

HARO

```
? Q("AnnIE").AtLeast(2).VowelNB()
#--> TRUE
? Q("AnnIE").AtMost(2).VowelNB()
#--> FALSE
? Why()
#--> no: expected atmost 2, found 3
```

✓ ran: printed what this block promises

The chain refuses what its subject cannot do. It speaks the same lexicon as the machine's direction: a French word works inside a chain. And a misspelled word is refused with the nearest word. This is "one lexicon, two directions" in three lines:

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'?
```

✓ ran: printed what this block promises

A chain branches on its own truth, and an ordinal is a word:

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
```

✓ ran: printed what this block promises

Questions work the same way. A force word, the question's first word (`What`, `HowMany` and `Is` in the block below), opens a frame, one word and one method. A tag question ends in "or not". What does the session keep when one answers yes and the other no? It keeps every answer and folds them, like a 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
```

✓ ran: printed what this block promises

The session kept `[ 1, 0 ]` and folded it: all yes is `FALSE`, any yes is `TRUE`.

THE MACHINE IS THE PARSER

Now the other direction. The sentence is a string, and the library reads it. The same verdict can come from a narration of questions and from a chain of truth. How sure is the chain of truth, and why? Run it:

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
```

✓ ran: printed what this block promises

It answers `certainly`, and says why: `yes: "haro" is lowercase and is latin and containing "r"`.

The whole paragraph is compiled into code before anything runs, and the code is readable. It is the library's own account of why the layer exists:

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()
```

✓ ran: printed what this block promises

Does the same meaning in Hausa land on the same code? Hausa is written here without the hooked letter, as the pack reads it today, for a reason that chapter 11 gives. Run it, and each program says back what it understood, in its own words:

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
```

✓ ran: printed what this block promises

The comparison answers `TRUE`: one code for two languages!

The sentence-dictionary is not the only vocabulary. Where does a word go that it does not hold? It is looked up in the library's own vocabulary, here the documentation tags of the uppercase verb. A value in quotes is data, and never a verb. An unknown word in a value position stays a literal:

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
```

✓ ran: printed what this block promises

The third line is also a limit: it printed `a-_-b`. The word `sep` was in no dictionary, and was used as the text to replace. A word it does not know is a literal.

Five languages as data. The dictionary is keyed by meaning, and each language's word lands on one id. Do a French, an Arabic and a Turkish word land on the same one? Run it:

HARO

```
? StzResolveSemanticInLang("fr", "doublons")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemanticInLang("ar", "تكرارات")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemanticInLang("tr", "tekrarları")
#--> METHOD_REMOVEDUPLICATES
? StzResolveSemantic("removeduplicates")
#--> METHOD_REMOVEDUPLICATES
```

✓ ran: printed what this block promises

The languages follow their own grammar, and they are not word-for-word translations. Turkish puts the verb last, with the parameters before it. Arabic, with its vowel marks and a doubled letter, is stripped by the pack; the suffix on the verb is the masculine "it" for a text. French elides:

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
```

✓ ran: printed what this block promises

Three languages, three grammars, one answer: `orah`. Arabic says back what it understood, in its own words: `haro -> اعكس أنشئ سلسلة بـــ`.

No native reader has read these sentences. They are modelled on the guard's, with 'haro' in place of the guard's own test word, and the Turkish string reversal is the book's variant of the guard's list reversal. I do not know that a speaker has read them. The record holds no reading by a speaker of the Arabic, Turkish, French or Swahili sentences, as it holds none for the Hausa of chapter 11. They are printed as sentences modelled on the library's own, unread.

Can a language be a block of data? Here is Swahili, added at runtime, by the guard's own words:

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
```

✓ ran: printed what this block promises

Four mappings, one phrase, and the sixth language answers `[ 9, 4 ]`!

Natural code is data too. A document's fenced blocks run, and a template is filled:

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 ]
```

✓ ran: printed what this block promises

What the packs can say. How much does each pack say? The library has a function for it, and no guard prints its figures. I count against twenty-five reference meanings, which are the structural ones and every method and modifier any shipped pack names. They are not the thousands of verbs the library grew:

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
```

✓ ran: printed what this block promises

The Hausa pack speaks nine of twenty-five meanings: `ha 9 of 25`! And no model is loaded. The layer has an optional rescue by embedding, when nothing lexical was found and a model is loaded. At this commit none is, and none is needed. So I

say "no model is loaded", and not "never touches a model".

FROM THE 2020 DIAGRAM TO THE THREE ERAS

The library names three eras, in a design document of 10 July 2026. The first is near-natural language, the first paradigm. The second is natural programming: lower the language toward code, with free text, and one object per block. The third, of 2026, is one lexicon shared by execution and documentation, multilingual packs with morphology, and `Understood()`. I call the third *the shared lexicon*. The vision's own heading for it uses a word this book does not use.

Here are the dates, as the record gives them. My strategic diagram carries the years 2020 to 2025, shared on 13 September 2026; its Narrations band reads "executable natural language". The deck of 2022 was exported on 20 September 2022, and its slide 2 says to express thinking in an executable natural language. The library's repository begins on 12 March 2022, and its history cannot date 2020 or 2021. The near-natural files and the engine at its present path were first committed between 16 and 19 September 2025. On 30 October 2025 languages became configuration. On 2 November 2025 string input replaced brace capture. On 9 July 2026 the lexicon was unified from the documentation tags. On 10 July 2026 the two design documents were written, and the audit of fifteen canonical examples found that four ran, and two of those answered correctly. The site's flagship page is of 8 October 2026. The git dates come from the library's own repository, which the snapshot lacks; I checked them there on 9 October 2026, and the commits cited are ancestors of the pin.

So the earliest slide the record dates is the founding slide of the 2022 deck, exported on 20 September 2022. The year 2020 is the first year of my strategic diagram, which was shared on 13 September 2026; the record holds no slide of 2020.

Now the decision, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I built it without an AI model, as a dictionary. The design says everything in the natural folder attacks one wall: the semantic gap between a human intention and its execution. The thesis of near-natural language, in that review: zero NLP, zero ML, native speed, full determinism, and the programmer in control. The natural vision's closing observation is that the layer delivered a fourth promise the old documents never dared to make: accountability.

I decided on two directions, one discipline. The review's table has five rows. Who parses: a machine with a lexicon, or the human. Ambiguity: refused loudly, or impossible by construction. Vocabulary: bounded by packs, or unbounded. Understanding: verified by the machine, or by a human reading. Audience: domain experts and AI agents, or programmers. Both end at the same place: a deterministic sequence of semantic operations.

I refused markers on reference. On 10 July 2026 I rejected the family of suffixes that marked a reference. The cipher was inverted by its own inventor from memory, it duplicated meanings, and language itself never marks reference this way. More than two hundred and sixty methods were removed. Chapter 11 tells it in two sentences.

I refused the ungrammatical chain. The chain is the sentence, and an ungrammatical chain that runs is a bug. I hold this as a view, not a result. The doctrine is held by reading aloud, not by a machine, and I show the proof below.

I refused negation as a prefix. Negation lives in negative forms, in antonym comparators and in the tag closer `OrNot()`. The tag question above is that ruling running.

I refused silent tolerance. The old typo tolerance was replaced by explaining everything: the refusal with the nearest word, above.

I refused global context. Two chains interleaved contaminated each other, and the fix put the context on the chain. The process-wide `Why()` remains as a convenience for the console and the tests. The globals survive as fallback shims: a legacy method outside the natural layer that builds a new object still falls back to them.

I refused the markup language. Annotations that handed the machine what it could not infer governed by making the human annotate. The system governs by dialogue. Two seeds were kept as thin doors, and they are the document and the template above.

I refused a command interpreter for narratives, and brace capture. In my own words in the article: it is designed for commands, not narratives. And one object per block. The string form replaced the brace form, because natural code became a first-class data type, storable, versionable and searchable. The rule since relaxed: a program may name objects and switch between them.

I refused governance inside the natural layer. I ruled that scope out of the library on 12 July 2026, and the code was removed.

I refused natural language as a way to declare. It asks, and never declares.

The court comes next, and one guard in it fails.

5. THE COURT

`nnl_narrated` proves the flagship sentence, the generated surface of nouns, the degree words, the conditional mood, the ordinal reference, the refusal that accounts for itself, the grammaticality doctrine, context on the chain, and the discourse tense. `multilingual_pack_narrated` proves packs as data, programs in French, Arabic and Turkish that run, verified affix stripping, a language added at runtime, and `Understood()` back in the program's language.

`question_narrated` proves the question frames and that a frame with no aspect is refused. `literate_templates_narrated`, the documents and templates. `truthchain_narrated` and `interrogative_narrated`, the folded verdicts.

`semantic_resolver_narrated` is the guard that ought to protect the dictionary, and at this commit it fails. I ran it twice on 8 October 2026 and each time it stopped with an error: three assertions failed, and the run ended after the twenty-ninth. In that run the word "caps", which the guard asserts, resolved to a method for bounds, and "mirrored" resolved to nothing. The reason is the design's own: the vocabulary is read from the library's documentation tags, plus seventeen hand-written operations grown by reflection to about three thousand. So a documentation edit changes what a word means. The library refuses honestly in the second case: in that run `Unresolved()` named the word and `UnderstoodAll()` printed 0. The court of the dictionary has failed on the dictionary, and chapter 11, chapter 22 and chapter 2 name this guard.

I ran the other guards named above once each on 8 October 2026, and in that run they passed all their assertions; the book's own data holds only their counts. The assertion calls written in the seven guards named above sum to 269.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in breadth of phrasing. An AI model accepts any phrasing in any language, and the dictionary accepts what the packs name. Measured above: the Hausa pack covers nine of twenty-five meanings, Turkish ten, Arabic sixteen, English twenty-three (the two it lacks, English reaches through the grown vocabulary), French twenty-five. And the sentence layer acts on three classes only: strings, lists and numbers. A customer's domain objects are not among them; the domain goes through the knowledge graph.

The field leads in whole-program semantics. Inform 7 compiles a whole world. Here a program is a sequence of operations on a live object.

The field leads in an editor. In my run, the suggestion function answered nine continuations for a half-written sentence, and the code credits the controlled-language field for the idea. There is no editor here that shows them as you type.

And there is a loss on the library's own side. `Understood()` prints verbs and not their arguments: the block of section 3 prints `contains` and not the 'r' it asked about. In my run, `Replace` printed as `replace`, and `Add 3 to it` printed as `add` while the result was 8. A reader who checks the machine's reading sees no 3. This qualifies chapter 11's claim that a person can check what the machine understood.

Where the win is, with the loss beside it: the same sentence gives the same answer, with a reason, in the user's language, and designed to run offline, which no guard measures. The loss is that the user's language must be in a pack, and a word outside is a literal or a refusal. The record says the winning case is regulated or sovereign work: an instruction that executes locally and can be audited to the word.

The next section says where the chapter stops.

7. THE LIMIT

BUILT	both directions: chains where the human is the parser, and sentences where the machine is; five language packs as data and a sixth added at runtime; refusals with reasons on both sides; question frames; a thin truth chain; natural code as data in documents and templates
DESIGNED	a grammar that holds an AI model to the lexicon; a guard that no socket opens during a natural run; the plan emitted as a chain; the article examples pinned as fixtures
ABSENT	any machine that judges a chain's grammar; Ajami, Zarma and Fulfulde; a natural layer closed all the way down; a dictionary protected from the drift of the documentation it is read from; the old articles' promises, several of which do not run

An ungrammatical chain runs. The second chain below reads "which a length of 4", which is not English. Does anything judge it? Run both:

HARO

```
? TheWordQ("haro").IsAQ(:Word).WhichQ().HasQ().ALengthQ().Of(4)
#--> TRUE
? TheWordQ("haro").IsAQ(:Word).WhichQ().ALengthQ().Of(4)
#--> TRUE
```

✓ ran: printed what this block promises

Both answer `TRUE`: nothing judged the second! The doctrine is that reviewing a chain is reading it aloud as English. It is the human's job.

Does a chain's descriptor reach host code? Yes. `IsAXT` builds a string and evaluates it, so a function defined by the program and named in the string printed `host code ran`. This is the residue that the study of the record and chapter 6 name:

HARO

```
? Q("haro").IsAXT([ "String('x') and HostRan() and @isString" ])
#--> host code ran
#    TRUE

func HostRan()
        ? "host code ran"
        return 1
```

✓ ran: printed what this block promises

And a word of the old era is reported, and never guessed. Permissive mode still runs the rest, and strict mode refuses. The suggestion offered, `addn`, is a poor one:

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'?)
```

✓ ran: printed what this block promises

Several chains of the old articles no longer run. The sentence of the bridging article that carries `IsAMQ` needs a marker that I removed by my ruling of 10 July 2026. Several chains of the near-natural article do not run at this commit: `InLowercase()` raises for lack of an object; `LettersNB`, `BeforeQ` and `TheirQ().FirstCharQ()` are not defined; and the brace form `Naturally() { ... }` raises for too few parameters. The bridging article's chain with `ReplaceLowercaseCharsQ` is not defined either. I ran twelve chains of the near-natural article on 8 October 2026, and eight work. In that run, the bridging article's paragraph still ran in permissive mode, and printed two nested boxes around "NOTHING SOFTANZA" instead of the one it promises. The machine said so, since `UnderstoodAll()` is false and the unresolved word is named (the block above shows both), but a reader has to ask.

What is built: both directions; five packs as data and a sixth at runtime; refusals with reasons on both sides; the question frames, the thin truth chain, the evidential register; natural code as data. The folder holds thirteen files and about twelve thousand lines in twelve classes, by `wc` and `grep` on 8 October 2026.

What is designed: the plan of a narration emitted as an executable chain, so that chains gain `Understood()` for free. A grammar emitted for the natural lexicon, so that an AI model may only produce sentences the layer executes: I searched the folder, and no grammar emitter touches it. A guard that fails if a socket opens during a natural run, which does not exist. A coverage figure per pack printed by a guard. And the articles' examples pinned as fixtures.

What is absent: Hausa in the Ajami script, Zarma and Fulfulde. A natural layer closed all the way down: a plain list in a natural sentence reaches host code, which chapter 11 prints, and a descriptor reaches host code, above. Any machine that judges a chain's grammar. The standard Hausa letter, which the pack stores broken. A dictionary protected from drift. The record also holds no account, and no date, of my plea to the project of the former host language that natural programming be made a first concern, nor a sentence of mine in that voice that the book can source. All it holds is a private memo of 30 August 2026, which gives my judgement of the language's shipped natural form. The memo is marked private, so the book does not quote it and sources no phrase of mine to it.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Write the chain as the sentence and read it aloud. The library refuses what its subject cannot do, and says why. You are the parser, so the sentence is only as grammatical as you are.
The analyst	A requirement can be a paragraph that runs, with a paraphrase that comes back. Today it acts on strings, lists and numbers, so the analyst's domain words go in a knowledge file (chapter 14), not here.
The designer	A designer's tone is in the refusals: a message names the word and the nearest word. The record has no designer-specific evidence for this chapter, and I keep the line short.
The decision-maker	Ask any "natural language" claim what it does with a word it does not know. Here it reports it, refuses it in strict mode, or takes it as a literal, and says which. It is designed to run offline with no key. The no-network claim is not yet measured.
The educator	A learner writes in Hausa, French, Arabic or Turkish, and the program says back what it understood in the same language. The Hausa pack speaks nine of twenty-five meanings, and no speaker of those languages has read this page's sentences, as far as the record shows.
The agent	Speak the lexicon. Use strict mode and an allow-list (chapter 11). Expect refusal with a reason. Your sentence is a proposal. A grammar that would hold you to the lexicon is designed and not built.

9. EXERCISE

Take one sentence you say about a word in your own language: that it is lowercase, has a given length, and a given number of vowels. Write it twice before you run anything. First as a near-natural chain, where you are the parser. Second as a `Naturally` sentence, where the library is. Under each, write the line you expect it to print.

HARO

```
# 1. as a chain, where you are the parser:
# 2. as a Naturally sentence, where the library is:
```

✎ not run: an exercise, written for you to run

Then run both. Then change one word of the chain, and one word of the sentence, to a word the dictionary does not hold. Which direction told you? Which did you have to tell yourself? In one direction the machine tells you, and in the other you must tell yourself. If a promise breaks, do not change it: explain the output first.

I left this one in your hands. Section 7 shows a chain that reads "which a length of 4", and nothing refused it. Run it, read it aloud as English, and write under it the line you would have the library print to refuse it, with the reason it would give.

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:
```

✎ not run: an exercise, written for you to run

When you have written that line, you will know why I hold the ungrammatical chain as a view and not as a result, and who the page says must judge it (sections 4 and 7).

The next chapter is about knowledge programming, where the analyst's domain words were sent. Its principle is that a system is a file, and that adding a line to it makes the system know more, with nothing coded or trained. Here a sentence asks, and never declares.

WHERE THE EVIDENCE LIVES

NARRATION	`stznatural-vs-ring-naturallib.md`
NARRATION	`stz-near-natural-language-programming.md`
NARRATION	`stznatural-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
ON THE SITE	natural
RULINGS OF THE STUDY	12.2, 12.6, 12.27, 12.30

PART III · CHAPTER 14

Knowledge programming

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 13 of 13 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the three practices of the field that meet in this idea, the semantic stack, the property-graph database and rules over facts, and set beside them a system that is a plain text file. I run that file: two facts, one law and a proof with its steps, then a clinic in strict mode, where a refused rumour is kept and named. I write the file, read it back and show it at scale, from a course catalogue to an AI agent's memory. I say what I decided, that a business describes itself in one file and that the file is the program. I say what I refused: a parallel world, a contradiction resolved in silence, retrieval. I print the limit, a file that at this commit has no rules. It stands beside the principle: a system is a file, and adding a line to it makes the system know more, and nothing is coded or trained.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Three practices of the field meet in this idea, and none of them is the same as it.

The first is the semantic stack: RDF, OWL and SPARQL. A fact is a triple of subject, predicate and object, written in a shared vocabulary. An ontology says what classes and properties exist and what follows from them, and a reasoner derives the consequences. These are W3C recommendations, of 2014, 2009 and 2013. What the field asks of the author of a knowledge base is an ontology language of their own, a store and a query language. They usually stand beside the application and not in it.

The second is the property-graph database, with its query language. The record has one comparison that sets such databases against this library. It is a text an AI model wrote, kept in the library's record, and it credits the field with scale, persistence and transactions.

The third is rules over facts, in Prolog and in Datalog. Facts and rules in one program, a query by resolution. The record names Prolog only, as a language to reach, and not to rebuild.

A fourth practice stands behind the contrast, and the design document names it only to refuse it. It is retrieval, a pile of document fragments searched by similarity; that description is general knowledge, not a study of the record.

So in the field the knowledge lives in a store and its logic in a reasoner, both beside the program. Here the knowledge is a plain text file that the program loads.

2. THE FIRST PRINCIPLE

A system is a file. Adding a line to it makes the system know more, and nothing is coded or trained.

Two sentences of the record carry it. The definition, in the design document written from my directive of 13 July 2026: intelligence is the computational ability to derive new knowledge from existing knowledge. And what the 2026 vision keeps from it: adding one fact augments the system's intelligence, with no code change and no training. Chapter 17 opens on the first, and I open here on the second. The design document of 13 July 2026 already says it, in nearly the same words, and the chapter keeps its opening.

Now the form itself, a text and what it does when you run it.

3. THE SOFTANZA FORM

Here is a system as a text. Two facts: flour is made from millet, and couscous from flour. Is couscous made from millet? Ask the graph, then add one line to the file, a law saying that *made from* is transitive, and ask again:

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
```

✓ ran: printed what this block promises

Without the law the graph answers `not derivable`; with it, `proved: couscous made-from flour made-from millet`. The text is the program. The same two facts, one extra line, a different capability. The block does not touch the disk: `ImportKnow` takes a text, and takes a path when the string ends in the file's extension.

Now add a fact to the graph with the law, and nothing else. What do the two graphs answer to the longer question? Run it:

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'
```

✓ ran: printed what this block promises

The graph with the law proves the longer chain and writes its steps out: `proved: porridge made-from couscous made-from flour made-from millet`. The graph without the law, which also never received the fact, still answers `not derivable: no recorded fact or lawful chain gives 'porridge' made-from 'millet'`, and I want you to notice that this no has two causes, named in its

own line: no recorded fact, and no lawful chain. A refusal is as certain as a proof, about what is recorded and declared: the graph says "not derivable", not "false". The library's comment on `Prove` says it: `a certain no is still certain`.

The same growth reaches the domain language model of chapter 19. Nothing is trained. But a model forged from a graph, in chapter 19's word, is a snapshot. Does the model learn the new fact when the graph does? Run it:

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.
```

✓ ran: printed what this block promises

It does not. The second answer is the old model still refusing, `That is outside the 'coop' domain`, and only the model forged again answers `Millet is a grain.`

The file the library writes. A clinic. A relation declared unique, and a graph in strict mode, where every fact needs its provenance. What does it do with a fact that has no source, with one that has, and with a second, contradicting one from a rumour? And what does it then write to the disk? Run it:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The naked fact is refused with the library's own sentence, `Strict mode: every fact needs provenance`. The fact with its source and confidence is admitted. The rumour is refused, `0`, and the refusal is kept! That is what the library writes, a header and up to four sections. They come in this order: the facts, the provenance when a fact carries any, the contradictions when one was refused, and the ontology when a law was declared. One line per record, fields separated by a bar. A bar or a line break inside a value is escaped. The refused rumour sits in the file, named, and never resolved. The code comment on the contradiction list says it: `named, NEVER silently resolved`.

Does the file read back into a strict graph, with the law, the source of every fact and the refusal inside it? Load it:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

It does: one fact merged, the source and the confidence back, `[ [ "source", "registry" ], [ "confidence", 0.95 ] ]`, one contradiction, and the law. A file of this kind was once written without its provenance. The export wrote only the bare triples, so knowledge read back from the disk no longer said where it came from. A strict graph could not load its own export. The fix is the two sections above. The guard that proves it is named in section 5.

A law that fires when a fact is added. On the library's default world, the one behind the natural-language sugar, a symmetric law derives the reverse fact as the first one is added, with no call to a derivation step. Nobody wrote that Ibrahim is married to Amina. Does the default world know it?

HARO

```
StzConstrainRelation("married-to", :Symmetric)
StzKnowRelation("amina", "married-to", "ibrahim")
```

```
? DefaultKnowledgeGraph().Query([ "ibrahim", "married-to", "amina" ])
#--> 1
```

✓ ran: printed what this block promises

It answers `1`. The reverse fact is there, and nobody wrote it! This is the only place where a declared law fires on add that a run in this book shows, and it is the default world. The graph class also has a switch, `EnableAutoDerive`, that runs the derivation rules after every edge added; it is off by default, and I did not run it. I show the same law on an instance graph in section 7, where it does not.

A system at scale. The education program of the library is a knowledge file. Its catalogue, `program.zknw`, says what the program is called and which courses and languages it has. What is its name, and which languages does it carry? Ask it:

HARO

```
oP = StzProgramQ("../../education/program")
? oP.Id()
#--> softanza-education
? @@( oP.Languages() )
#--> [ "en", "fr", "ar", "ha" ]
```

✓ ran: printed what this block promises

It is called `softanza-education`, and it has four languages. Adding a course is adding a line there, and the guard checks that every declared course is a folder and every folder is declared. In the education module the snapshot holds ninety-nine knowledge files, counted with `find` on 8 October 2026. They are courses, exercises, worlds and skills, a project world for a learner, and the overlays an institution lays over the core. An agent's memory is also a knowledge graph, saved through the same format, with provenance.

That is the form. Now what I decided, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I decided that a business describes itself in one file, and that the file is the program. The design document records my directive: entities, relations, an ontology of domain-specific semantics, and rules, in a file that is readable, diffable and hand-editable, validated on load. The last two words I have not yet met, and section 7 says where.

I decided on four-letter extensions. `.zknw` for knowledge, and the others of the family; the old spellings still read. The snapshot writes `.zknw` and still reads `.stzknow`. The file of rules was not renamed: the graph still writes and reads `.stzrulz`, though the family names four letters for it. I report it as the record has it.

I refused a parallel world. The natural layer once held its entities in globals, and the design document called it the wrong home, the wrong shape and a parallel world. The resolution is one default graph standing behind the natural-language sugar. The old globals still exist, and the graph grows beside them: the library's header calls it a dual write. The law behind it is that naked domain globals are forbidden; a global form may exist only as thin sugar over one default instance. So the dialog of the symmetric law above is sugar over a graph of the same class as the first blocks used.

I refused to resolve a contradiction in silence. Strict mode, where every fact needs its provenance, is opt-in knowledge hygiene.

I refused retrieval. The design document says: no code change, no training, no RAG machinery. It is the same refusal as chapter 17's.

5. THE COURT

`knowledge_integration_narrated` proves that the world answers `WhatIs`, that the graph grew alongside it, and that the law lives in the ontology. It proves that `Prove` narrates the law first and then the links of the chain, with certainty 1, and refuses what it cannot prove. It proves that the file round-trips. It proves that a symmetric law derives the reverse edge when the derivation step is applied after a fact is added through the graph. Last, it proves that strict mode admits a provenanced fact, refuses a naked one, and refuses and records a unique contradiction.

`provenance_survives_save_narrated` proves that a fact's source and confidence survive a save, that a strict graph loads its own export, and that a refused contradiction survives. It proves that a bar or a line break in a value is data, and that an old file with no provenance still loads. It proves that an agent's memory records who learned a fact, and when.

`program_narrated` proves that the program counts its courses, and that every course declared is a folder and every folder is declared. It proves that the texts read the same from a checkout with other line endings.

That is what the guards prove. What the field does better comes next.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The record's own graph comparison says where the traditional solutions win: large-scale systems that need distributed storage and processing, and performance-critical applications. Its table puts Softanza at business scale, in the hundreds to thousands of nodes, in a single process. That is the fair summary of the loss: the field leads in scale, persistence, transactions and tooling. The comparison's row for persistence predates the file; the knowledge graph now writes one.

The field leads in standards and reach, in general knowledge. A knowledge base in RDF can be shared with any tool that reads the W3C formats, linked to public vocabularies and queried across stores. As far as the record shows, a knowledge file is read by this library and by nothing else. A reasoner for a rich ontology language does what three laws and a class declaration do not. The field leads in the expressiveness of rules: Prolog and Datalog derive new facts from arbitrary rules. Here, in the runs I kept, one law derives a fact, and only on the default world. The field leads in multi-step queries, since SPARQL and Cypher join patterns. And in many relations between two things.

Here is where I think this stands apart, with the measured limits beside it. The knowledge is a text file a person can read, edit, diff and ship with the program. Every answer is provable with its steps, and a no is as certain as a yes. What a fact claims travels with its source and its refusals. And the same format carries a course, a learner's project and an agent's memory. None of that needs a store, a server, an AI model or a training run.

The losses are printed. The limit of the chapter comes next, with its runs.

7. THE LIMIT

BUILT — a system as a plain text file of facts, with laws (unique, transitive, symmetric) and the source of each fact; Prove with its steps, a no as certain as a yes; strict mode; the file written, read back and shipped, from a course catalogue to an AI agent's memory

DESIGNED — rules in the file; derivation on any graph; a query that joins; several relations between two things; class inheritance; validation on load

ABSENT — the 2022 founding dialog, which does not run; Explain on a knowledge graph; the rules section of the file, which the parser drops

The limit of this chapter is the one my directive, in section 4, already carries: *entities, relations, an ontology, rules*. The file at this commit has no rules. Here is the file with a rules section. What does the parser make of it?

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
```

✓ ran: printed what this block promises

The file merges, `1`, and the ontology is empty, `0`. The parser recognises the header `rules`, and has a handler for none of the lines under it. They are dropped: the ontology holds `0`, and no refusal comes back. The same law under the header `ontology` gives an ontology of length 1. Rules live elsewhere, as data in a rule attached to a graph, which are chapters 29 and 30, and in a separate file that names rule functions. In this file there are facts and laws. I print "facts and laws", and I send the rules to those two chapters. The single file that holds rules, which the design promises, is not there, and this section prints the gap as the chapter's limit.

The law is also not derived on an instance graph. Take the same symmetric law and the same fact as on the default world. Is the reverse there?

HARO

```
StzKnowledgeGraphQ("pairs") {
        ConstrainRelation("pairs-with", :Symmetric)
        KnowRelation("margherita", "pairs-with", "chianti")
        ? RelationHasLaw("pairs-with", "symmetric")
        #--> 1
        ? Query([ "chianti", "pairs-with", "margherita" ])
        #--> 0
}
```

✓ ran: printed what this block promises

The law is recorded, `1`, and the reverse edge is not there, `0`! The class header says it: `only :unique and :transitive are acted on here`; symmetric is only recorded. Four more limits, each with its run.

One relation between two things. The graph stores one edge per node pair. Does it admit a second relation between the same two entities, `amina` and `bread`? Run it:

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).
```

✓ ran: printed what this block promises

The second relation is refused, `0`, with the library's own sentence, `Can't add fact: 'amina' and 'bread' already carry the edge 'sells' (stzGraph stores one edge per node pair)`. A knowledge model that cannot hold *sells* and *buys* between the same two entities is a real limit. The code says it: `known limitation, revisit with multi-edge support`.

A two-step question answers only its first step. Does `QueryPath` follow its second pattern, and can the predicate of a query be a variable? Run it:

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

✓ ran: printed what this block promises

The path answers `[ "couscous" ]`, the first step and nothing more: `QueryPath` ignores the rest of its patterns, so it is no multi-step query yet. The variable predicate answers `[ ]`.

A line that is not a fact. Four lines, three of them not facts: two fields, four fields, and no bars. What does the parser refuse?

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])
#--> [ ]
```

✓ ran: printed what this block promises

Nothing. The file merges, `1`, and the refused list is `[ ]`; the parser is silent and the three lines vanish! The design document promises that the file is "validated on load". At this commit the parser keeps only a line with exactly three fields.

The 2022 founding dialog. The narration tells knowledge programming through a dialog: an apple as a fruit and as a company, an owner, relations asked as questions. At this commit its first line does not run. The text, which I show and do not run, is `@Q("Apple").IsA(:Fruit)`. On 8 October 2026 it stopped with `Calling Function without definition: @isfruit`; I did not keep that run as a block. The string class reads the predicate as natural language and looks for a function by that name. Two other forms stop elsewhere. The narration's promise lines are stale. The nearest form that runs keeps the point. Does the world know one name with two meanings?

HARO

```
StzKnow(:Apple, :Company)
StzKnow(:Apple, :Fruit)
? @@( WhatIs(:Apple) )
#--> [ "company", "fruit" ]
```

✓ ran: printed what this block promises

It knows both: `[ "company", "fruit" ]`.

The study of the record already says the dialog was kept at the margin, and that the centre moved to the file. The chapter keeps the file as its opening, and the 2022 dialog stays at the margin.

What is designed: rules in the file; derivation on any graph; a multi-step query; several relations between two things; class inheritance. `DefineClass` writes a fact, and no inheritance follows. The nearest that ran, in an earlier probe whose run I did not keep, is to declare `is-a` transitive and state the class as a fact of that relation. And the design's knowledge hygiene beyond provenance, bounded queries, scoped graphs and revision with rollback, has no method of that name.

What is absent or broken: `Explain()` on a knowledge graph, which calls `ApplyInference`, a function defined nowhere. And `ValidateOntology()`, which returns 1 whatever the ontology holds, as its own comment says, `known defect`.

One note on the design document's history. Its amendment of 22 August 2026 says that no knowledge file exists anywhere in the tree, and that this is hygiene and not a gap. At this commit that is no longer true: there are ninety-nine, all of them in the education module.

That is the limit, printed. Now the six doors.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	One class and a text file: `Know`, `KnowRelation`, `ConstrainRelation`, `Prove`, `Query`, `WriteToKnowFile`, `ImportKnow`. The file is diffable and ships with the program.
The analyst	The offering page names it: what you declare is your organisation's entities, rules and flows, in a plain-text knowledge file. Read "rules" as the limit above. Where to start, in the course: the chapter called "Teach a world".
The designer	The record is silent. Neither the offering page nor the study ties the knowledge file to the designer's door, and I do not invent one.
The decision-maker	The file is plain text that a manager can read. Every fact can carry a source and a confidence. A refused claim is kept, named and never resolved. An answer comes with its proof, and a no is certain. The design's second law says no capability may require anything paid or remote. I did not run the file offline.
The educator	The whole education program is knowledge files. A learner teaches a world, then questions it, and the project's guard checks the world file, the agent file and one printed answer. An institution adapts by overlay, never by fork.
The agent	Your memory is a knowledge graph with provenance, and your world is a file you load and query.

9. EXERCISE

Here is a world with two facts, and a question it cannot answer yet. Your promise is the line proving that class 3b is part of primary, with the chain written out. Change the world, and not the 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.
```

✎ not run: an exercise, written for you to run

Run it and see what prints. If you change the question to ask about grade 3, it will prove something, and it will not be the line you promised. Then ask the reverse question, whether primary is part of class 3b, once your world has its law, and see whether the law adds a fact in the wrong direction. I do not tell you the line to add.

I left this one in your hands. Section 7 runs a file of four lines, three of them not facts, and prints what the parser says about them. Here is that file again with one change: it also asks the graph about three of the lines, and I print none of the answers.

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" ])
```

✎ not run: an exercise, written for you to run

Run it. Read `merged`, read `refused`, read the three queries, and find which lines the graph holds and what the parser told you about the others. Set the design document's promise of a file "validated on load", in section 4, beside what you find; section 7 says what the parser keeps. I do not give you the answer, and I did not fix the parser. When you have it, you will know why a file that merges is not a file that was validated.

The next chapter is about pattern languages: a pattern is a rule about the shape of data, written as data, and the same habit serves data of every kind. Here a law about a relation was a line in a file; there the rule is about the shape of the data itself.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-knowledge-programming-narration.md`
NARRATION	`stz-bridging-minds-and-code.md`
NARRATION	`stz-rules-as-graph-rules-narration.md`
ON THE SITE	offering
RULINGS OF THE STUDY	12.2, 12.6, 12.12, 12.13, 12.19

PART III · CHAPTER 15

Pattern languages

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 13 of 13 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I start from the regular expression, which is what a pattern is for a working programmer, and carry its habit to data that is not text: lists, numbers, tables, matrices, time and graphs. I run the same small move in each medium, a constructor, a pattern in a string, `Match`, and I show you at once that the grammar is not quite one. I say what the guards prove and where the family is still in development, and I keep its one tension for the limit: a word the language does not know is not yet refused. The principle is this: a pattern is a rule about the shape of data, written as data, and the same habit serves every kind of data.

1. THE CONCEPT, AS THE FIELD HOLDS IT

For a working programmer, a pattern is a regular expression: a compact notation for the shape of a string, handed to an engine that says whether the text fits, and where. The engines are old and well known. The one beneath this library is PCRE2, and the site measures it against two others.

For data that is not text, the field uses other tools, one per medium and none shared. Structural pattern matching in the languages of the ML family, in Rust, and since Python 3.10 in Python. Schema checks for lists and records. Data-quality libraries for tables. Row pattern recognition in SQL, and stream engines, for sequences over time. And graph query languages, whose core is a path pattern. That is general knowledge, well established; the record holds no dated study of any of them, and I invent none.

The estate's own field study of August 2026 gives the small languages that survived a profile: a closed grammar, one domain question, plain text, a tiny surface, a specification with a conformance suite, decades of stability. A pattern language is exactly that. One question, does this data fit, in a few characters. What the field does not offer is the same notation across the media.

Next, the first principle.

2. THE FIRST PRINCIPLE

A pattern is a rule about the shape of data, written as data. The same habit serves every kind of data.

The sentence the library's articles return to, in each medium, is the one in the article on numbers: instead of instructing the machine how to test a condition, you state what you mean. My study of the record, which is not the field study above, also carries a phrase of mine, "for the first time in computer science history". I found it in no text of the library, so the site does not claim it. I keep it here as what I said, and I claim only what is measured: one regex-shaped grammar across seven kinds of data in one library, text and six that are not text, with the same small-function habit. And I will show you at once that the grammar is not quite one.

3. THE SOFTANZA FORM

A pattern called by name is a closed catalogue. The library has a data file of named patterns. What does it say when you ask for a name it does not hold? Run it:

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.
```

✓ ran: printed what this block promises

It refuses, and says why: `The pattern name you provided does not exist in stzRegexData file.`

A pattern can also answer the half-written value, which is what a form wants as the person types:

HARO

```
StzRegexQ("^\d{3}-\d{2}-\d{4}$") {
        ? MatchAsYouType("123-45")
        #--> 1
        ? MatchAsYouType("abc")
        #--> 0
        ? IsPartialMatch("123-45")
        #--> 1
        ? IsPartialMatch("123-45-6789")
        #--> 0
}
```

✓ ran: printed what this block promises

Now the move that gives the chapter its title. Take a list. Its pattern is written in square brackets, and counts, nesting and negation are written, not looped. `@N` stands for a number, `@S` for a string, `+` for one or more, `@N1-3` for one to three, and `@!` for not. Does `[@N1-3, @S]` accept four numbers before the string? The second call answers:

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
```

✓ ran: printed what this block promises

No: the second call prints 0, because four numbers are one too many. Every call here uses a fresh object, for a reason that section 7 gives. Chapter 11 showed four calls of the same language, and I do not repeat them.

The numbers have properties, and a property is meaning, not procedure. Which numbers between 1 and 500 are perfect? The last line turns a pattern from a test into a search:

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 ]
```

✓ ran: printed what this block promises

Three numbers answer, and no loop was written.

Tables: a data-quality rule is a sentence of the language. The person who owns the data can read it. Niamey appears twice, so the cities are not unique, and the negation of that is true:

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
```

✓ ran: printed what this block promises

A pattern is data, so patterns compose: one is built from patterns.

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
```

✓ ran: printed what this block promises

A matrix: a concept, square and symmetric, is one line.

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
```

✓ ran: printed what this block promises

Time, in the part of it that runs, written here with the class rather than its small function `Tmx`. A duration must fit a rule: one to two hours, on a step of a quarter of an hour. Which of three durations fits: ninety minutes, eighty minutes or three hours?

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
}
```

✓ ran: printed what this block promises

Only ninety minutes fits: eighty is inside the range and off the step, and three hours is outside.

A graph: a pattern reads like the path it looks for. The labels match without regard to case. Does direction matter? The second line answers:

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
```

✓ ran: printed what this block promises

Yes: the path from `served` to `placed` finds nothing, and the line prints 0.

Last, the habit. The pattern is data in every language, and a language that exposes its tokens can be read back by a person and by an AI agent. Five of the six that are not text do; here are three of them:

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
```

✓ ran: printed what this block promises

Graphs have no `Tokens`, no `Pattern` and no working `Explain` at this commit, and no small function either. There is no `Gx`, and there is no `Tx` for time, although the study's table names both. The small functions are `rx`, `Lx`, `Nx`, `Tbx`, `Mx` and `Tmx`.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I decided that a pattern states a meaning and not a procedure. That decision is in the articles, and the articles are mine. The same ones decided to carry the regex idea to non-text data with one habit: braces, an `@` for a token, `&`, `|`, `@!` for not, `@cs:` for case. Lists keep square brackets. The other five use braces. That is the first honest correction of the phrase "one grammar".

I refused to reinvent the notation for text. Simple matches are kept: a classic pattern string, named groups and captures work as written. Clearer names are put around it. The older work on naming is kept as history, for educational value. I hold that to be the right way to retire a mechanism.

I refused, for now, to build the siblings. The articles planned classes in which patterns are genetic, linguistic or quantic. The article says that they are in an early prototype and not in the library, and a search of the regex folder for "induce", "from example" and "infer" finds nothing. The study dates the finding of their purpose two years later, in the answer protocol (the way a person may answer by examples): induction from examples, and superposed candidates. That is chapter 18's ground, and chapter 9's refusal.

I refused the declarative builder as a headline. The builders are the classes that write a regex from named steps in code, not from a string. Eighteen of the fifty-one files in the builder's test folder are marked retired: the parser of a declarative pattern is pending. The article's lead example, `CanContainAChar(:Between = ...)`, raises. What runs is the conditional builder, and the look-around object. I do not build the chapter on the builder's headline form.

I have not yet refused a word the language does not know. That is the tension of this chapter, and I keep it for section 7.

5. THE COURT

`00_regex_narrated` proves the anchors and the search, the classes, the exact quantifiers, the groups, the four match types, partial matching as you type, and positions as data. `00_numbrex_narrated` proves prime, perfect, Fibonacci, even and odd, square and composite, each with a number that matches and one that does not. Its header records a regression in which `Match()` once returned TRUE for every input. `tablex_narrated` proves the parser, counting, uniqueness, sorting, types, sums and averages, the three connectives, sequences, and building patterns from patterns. Its header names the fault it killed. The parser sliced with an end position through a function that counts, and about ninety patterns parsed into a token of the right type with no value. `00_matrex_narrated` proves exact size, any size and the square shape. Its header records that `Match()` once returned FALSE for every input. `00_graphex_narrated` proves that a pattern compiles to a graph, present and absent labels, a sequence as a subsequence, the case rules, negation, alternation, numeric properties, and the cache by graph signature.

Note the placement: the guard for tables lives in the folder for regex, and its stem has no `00_` prefix.

There is no narrated guard for lists, time, or the regex builders. The courts of those are classic files without usable promises. The tests of lists sit after a function definition, so the host language reads them as the body of the function and runs none of them. I ran a copy with the function moved to the end, and I did not keep that run as a block. Sixty checks ran, and two of them disagree with the file's own expectation. The classic file of time says that a promise is TRUE and the run prints 0, and the central record filed this on 1 October 2026 as still open.

Next, the field, and where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

Strings. The field leads on safety and speed, and Softanza leads on naming. The site's page for the area says it itself: it trails the modern engines on safety and throughput, with no JIT and no linear-time matcher. The control against catastrophic backtracking is not reachable from Haro, although the code for both is already compiled into the binary. The engine's header confirms that the hardening exists underneath: a match limit, a recursion limit, length caps.

Lists, numbers, time, graphs. The field has older, more established single tools. The study says it without ranking them: one tool has list patterns, uniqueness and stepped ranges; three languages have structural matching; graph query languages exist; event processing exists for time. What the field lacks, and what the record claims, is the one habit across the media.

Tables. Data-quality libraries lead in reach and ecosystem. Here a rule is a sentence the person who owns the data can read. The loss is that it runs only on `stzTable`.

Where the family is still in development. The class headers print their known gaps as warnings: for matrices and graphs. That is rare, and I say it. Graphs: the quantifiers are parsed but not yet used by the match. Matrices: known gaps today, each carried as a warning on its method.

A date for the family's one recurring bug. The same root cause hit the family language by language: the parsers sliced with an end position through a function that counts. The numbers and matrices were repaired on 14 June 2026, the tables on 5 October, and the graphs in a fix whose date the file does not give. Time still carries it. This is evidence that the family shares one design and one parser's habits; it is also a likely reason time does not yet match.

Next, the limit.

7. THE LIMIT

BUILT	patterns for text, numbers and tables, each with a guard; the same small habit across the media (a constructor, a pattern in a string, Match, Tokens)
DESIGNED	one pattern engine shared by every medium; the pattern as the condition side of the cascade; the pattern written from examples
ABSENT	the refusal of a word the pattern language does not know, in the six media that are not text; sequences over a timeline in time; a narration and a small function for graphs; the genetic, linguistic and quantic siblings

What is built: strings, numbers and tables, each with a guard and a small function. The shared habit.

What is built in part: lists, matrices, time and graphs, in that order.

- **Lists**: types, counts, nesting, negation, alternation and sets run. Stepped ranges raise. In a run I did not keep as a block, the uniqueness flag was ignored, so `[@N2-3{1; 2; 3}U]` accepted `[1, 1]`, though the class has code for a uniqueness check. The wildcard of the article, `@A`, is not a token; the real wildcard is `@$`. And a reused object answers from a cache whose key is the list's length and the sum of its characters' codes, which does not depend on order. A fresh object answers 0 for `["hello", 1]`, and the same object after `[1, "hello"]` answers 1.
- **Matrices**: size, shape, element and three properties run. The class header lists the terms that accept every matrix: `row`, `col`, `pattern`, `determinant` and `sum`. A matrix of two by two is accepted by `{row(99)}`.
- **Time**: instants and durations with a step run. In a run I did not keep as a block, a range without a step raised. And sequences over a timeline do not run at all: here is a timeline with a meeting, a break and a lunch, and the pattern the article promises. Does it match?

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)"
```

✓ ran: printed what this block promises

The article promises that the first is TRUE. The second shows that even a single event the timeline holds does not match! The third shows why: the token's label is `"Lunch)"`, read with its closing parenthesis. That is the shared bug; repairing it in

time is the next step.

- **Graphs**: labels in order along a path, negation, alternation and numeric properties run. The arrow means *later on one path*, and not *next*. The quantifiers (`+`, `*`, `?` or a number after a token) are read and ignored. The graph of section 3 has three nodes; here `@Node10`, a node with the quantifier ten, is asked for between them. Does the pattern match?

HARO

```
? (new stzGraphex("{@Node(placed) -> @Node(served)}", g)).Match(g)
#--> 1
? (new stzGraphex("{@Node(placed) -> @Node10 -> @Node(served)}", g)).Match(g)
#--> 1
```

✓ ran: printed what this block promises

Cooking lies between the first and the last, so the first answer is the arrow's meaning. The second answers 1 too! The quantifier is ignored. The cache of this class is keyed by the node and edge counts and the node ids. A second graph with the same ids and other labels is answered from the cache, until `ClearCache()`: the answer can be stale.

The refusal the family has not yet made. Chapter 11's law is that a language is declared and closed, with refusals. What does the family answer when it is asked a word it does not know? Run it:

HARO

```
? Nx("{@Banana(3)}").Match(7)
#--> 1
? Nx("{@Banana(3)}").Match(1000003)
#--> 1
? @@( Nx("{@Banana(3)}").Tokens() )
#--> [ ]
? Nx("{@Property(Banana)}").Match(7)
#--> 0
```

✓ ran: printed what this block promises

A pattern with a word the language does not know matches every number! The tokens are `[ ]`, and nothing is raised. The matrix and table languages return an ERROR token that nothing raises, and the number and time languages return an empty list. The Haro charter session filed this as a finding, which the central record relayed on 28 September 2026: the six pattern classes accept unknown tokens silently. I reproduced it on this commit, and the request is not closed. A closed language that cannot refuse is not yet the law of chapter 11. The chapter prints it as a finding of the court, and not as a debt of mine. I hold that as my view. The run above, on this commit, is what stands behind it.

What is designed: one pattern engine shared by all the media. The engine design says the backtracking engine, the constraint solver and the pattern compiler are the same across all domains, and that only the vocabulary changes. Its status is planned. At this commit the vocabulary changes and so does the code. The engine's own file named *pattern* is another thing: palindromes, repeats and arithmetic and geometric sequences. The pattern as the condition side of the cascade, many faculties firing one after another on a shared state, is chapter 9's third rung, and designed.

What is absent: induction from examples, which is reserved. The genetic, linguistic and quantic siblings, which are vision only. The explanation of an unnamed pattern, since `rx("[a-z]+").Explain()` raises because the analyser class was never built. And any use of the pattern classes by the rest of the library: outside the regex folder, the tests and the docs, no class takes a pattern. `stzList`, `stzNumber`, `stzTable`, `stzMatrix` and `stzGraph` have no `Matches(pattern)`. In the docs, the education course uses them, in a chapter and an exercise whose answer is checked by running the learner's program, and the gallery has a few scripts that use them.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	One habit across seven kinds of data, with graphs still lacking the small function: a small constructor, a pattern in a string, `Match`. The pattern is data you can read and compose. Use the ones with a guard: text, numbers, tables, matrices and graphs, the last two with the gaps of section 7, and build nothing on the cache of a reused list pattern.
The analyst	A data-quality rule is a sentence of the table language. The person who owns the data can read it.
The designer	A pattern states a shape the interface can show before the data arrives, and a half-typed value is answered on the way.
The decision-maker	A rule is a line and not a procedure. It can be reviewed, and its verdict is 1 or 0. Say beside it that a pattern with an unknown word is not refused, and that a graph arrow means *later*.
The educator	The course teaches patterns in two chapters, in English, French, Arabic and Hausa. The ladder is a curriculum: text, list, number.
The agent	A pattern is a small closed notation that an agent can be held to and can read back. Be told that none of the six media that are not text can yet refuse a word it does not know, and that writing a pattern from examples is reserved.

9. EXERCISE

Three small tables are given. Write one pattern that accepts only the first. Before you run anything, write under each of the three printed lines what you expect and why.

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
```

✎ not run: an exercise, written for you to run

Run it. One clause rejects the second table and another rejects the third; which is which? And what would the pattern say if the email were a single space? I do not answer first. If a promise breaks, do not change it: explain the output.

I left this one in your hands, on purpose, and I name it a defect. Take the matrix block of section 3 and change one thing in its last line: put `{row(99)}` where `{property(identity)}` stands, and keep the matrix.

HARO

```
? Mx("{row(99)}").Match([ [1,0], [0,1] ])
```

✎ not run: an exercise, written for you to run

Say what you expect before you run it, then run it. Do not change the pattern to make the output agree with you: section 7 says where the library prints this defect, and why. When you have found it, you will know which terms of the matrix language accept every matrix.

The next chapter is about seven languages and one door. A door is a runtime and never a syntax, and the door is one: it is the answer to "what about language X".

WHERE THE EVIDENCE LIVES

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
ON THE SITE	languages
RULINGS OF THE STUDY	12.4, 12.18, 12.26

PART III · CHAPTER 16

Seven languages, one door

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 10 of 10 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the field's two lessons about reaching another language, an escape that is open and a border with no way out, and I put the door where they meet. I ask the door which languages it knows, run Python and NodeJS through it, read Python and JavaScript into one code graph, and show the governance refusing an executor with no posture. Then I say what I decided: one door, and no further host embeddings. And I say where the door stands at this commit: its governance is designed, a way back that is closed is absent, and the R door, in development, raises today. The principle is the one chapter 11 promised: a facet is a syntax of one grammar and never a runtime, a door is a runtime and never a syntax, and the door is one.

1. THE CONCEPT, AS THE FIELD HOLDS IT

For a working programmer, reaching another language from the one you write means one of two things. The first is a foreign function interface, a call into a shared library. The second is a child process, a run of an interpreter whose output you parse. Notebooks and polyglot virtual machines make the same move with more comfort. That is general knowledge; the record holds no dated study of these tools, and says so itself.

The estate's own field study of August 2026 has two sentences on this, and they pull in opposite directions. On the way out, the host language is the escape hatch, and the escape hatch is always taken. It gives two internal languages that died of

openness. On the way in, the differentiator is escape; a language you make must be consumable from any stack. It gives one language that stalled at the border of its own world.

So the lesson of the field is double. An escape that is open destroys a closed language. And a language with no way out dies at its own border. A door is where those two lessons meet.

There is a second habit, which looks like the first and is not. Tools that let you write in the syntax you know over another engine, a transpiler, a second front end for one compiler, I will call by the word of this book, *facets*, which chapter 12 defines. The estate's design memo for them is blunt about their cost. The promise of a second facet is readability for a human who knows the other language, and not behavioural transfer.

2. THE FIRST PRINCIPLE

A facet is a syntax of one grammar, and never a runtime. A door is a runtime, and never a syntax. The door is one, and it is the answer to "what about language X".

The vision's own sentence, ratified in session on 28 August 2026, says it of the door: foreign code in the selected languages, governed, one door, with no further host embeddings. The study of the record reads the pair in a line I keep: the facet brings the programmer, and the door brings the ecosystem. Chapter 11 promised that this chapter would tell the difference; chapter 1 promised it the seven layers. Here is the difference.

3. THE SOFTANZA FORM

The door's list is closed. The foreign languages are a registry, and the host is not on it. Which languages does the door know? Ask it about six, then about the host, under the name the library's own data still gives it, the former host language's, then about one that was never on the list:

```
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
```

✓ ran: printed what this block promises

"What about language X" has one answer, and for a language not on the list it is a refusal by name:

HARO

```
try
        oNo = new stzExterCode(:cobol)
catch
        ? cCatchError
done
#--> Language 'cobol' is not supported
```

✓ ran: printed what this block promises

The door refused it by name: `Language 'cobol' is not supported`!

Now the door runs a real runtime. A short note on the machine first. The door writes a script, starts the runtime by it, and reads a result file back. The library's default place for Python is not where this machine keeps it, so the door is given the bare command `python`, which the machine resolves on its path. Node and R are at the library's default places. The blocks that start a runtime are marked, in the reader, as touching the system, and they show the output that was recorded on the date of the build.

What comes home from a Python call? The result arrives as the host's own structure, a list, and the call leaves a 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
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The Python float `3.0` came back as `3`, in `[ "mean", 3 ]`. The trace also holds a duration. The block prints only `1`, the answer to `isNumber`, because a duration is a clock, and a clock cannot be promised.

That is the rationale of the whole plane, in one picture. Each layer of a solution is in the language best at it, and the host passes data in as text and reads data back. The library's own article lists the layers: system performance in C, event-driven systems in NodeJS, scientific computing in Julia, statistics in R, logical reasoning in Prolog, and machine learning in Python. It adds AI reasoning in a language model, and solution integration in the host. Here are two of them in a pipeline, Python computing the averages of two students and NodeJS deciding. The thresholds are mine; the data is the article's:

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" ] ]
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The data goes in by composing the code string with the host's own list printer, since the article's marshalling functions are absent (section 7).

Which seven? The title commits to a number, and I ruled on 12 October 2026 which one: the six foreign runtimes of the registry above and the host, as the article on the plane and the article on performance count them. The language model is not one of them, because it is not a door of this plane at all. Chapters 17 and 20 make the model a proposer that cannot commit, while the narrations run it as a first-class runtime through a function that is absent. Section 7 prints that as a limit. The record counts it differently elsewhere: seven domain modules with one

for a language model, seven domains with the host as an eighth, the registry's six without the host, the vision's "7 selected languages" that it never lists, and the engine design's five with COBOL. The registry above is the one that runs.

A facet is not a door. Reading a foreign language is also not running it. The two languages that I name as Haro's planned facets are also the two that the library can already read into one code graph. It does not run them and does not write them. One set of questions answers over both:

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" ]
}
```

✓ ran: printed what this block promises

This is a parser into an index of classes, methods and inheritance, and not a program graph that can print a program. It is a coincidence of the order of work, and not yet a facet. The record has one statement of mine on facets. I made it in session on 8 October 2026, and it says that a link should be made with the future

vision of Haro and its facet languages. In that vision the former host language is already there, and Python and JavaScript are planned. The charter of Haro on this machine speaks of one grammar and one text. Its design memo for a second facet, which proposes and rules nothing, recommends printers read by humans before any parser. Its own table lists the collisions: a Python reader expects zero-based indexing and Haro's is one-based. The chapter says "planned" only with that statement named as its source.

A door is governed. The governance has the word for foreign code. `external` is the posture the library defines as out-of-process. What does the governance answer before that posture is declared, and after? Ask it:

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
}
```

✓ ran: printed what this block promises

The refusal says why: `refused: 'python' has NO declared trust posture`. An executor with no declared trust posture never proceeds.

And the twin can rehearse a spawn before it runs, flagged as a risk. Chapter 24 owns the twin, and one block here is enough to say what the door does not use:

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" ] ]
```

✓ ran: printed what this block promises

Nothing is run there, and the plan flags the spawn: `RUNS an external command (side effects): spawn process: python analyse.py`. The script named does not exist.

Next, what I decided about the door, and what I refused.

4. THE DECISION, AND WHAT I REFUSED

I decided on one door. Its vision name is stzx, the door of the foreign languages, the x standing for external. The retired names are two older ones. The letter X was refused for algorithmics, because it would have sat beside the door, the clash that the vision's own file on the platform's letters named as its sharpest. The door took a letter and kept it.

I refused further host embeddings. The door is the answer to "what about language X". Chapter 1 promised this chapter on those words: seven languages were chosen for seven layers of a solution.

I refused to surrender the domains to generated code. The article on the domain library puts it as a stance. Rather than forcing programmers to master each domain, or to surrender to AI-generated code, provide carefully crafted abstractions. The site, as I read it on 9 October 2026, says the article has been in the repository since 23 April 2025. Its code is absent.

I refused a facet as a runtime. The study's plan says it in a line: a facet is a syntax of one grammar and never a runtime. So the door stays the only way to foreign code. It is a plan, proposed. It is not ruled.

I refused an approximating printer. The design memo's rule for facets: a printer refuses before it approximates. A facet spells the graph and never gives it the meaning of Python or JavaScript.

Now the court, and what it proves of the door.

5. THE COURT

`51_extercode_narrated` proves that the registry knows Python, R, Julia, Prolog, C and NodeJS, and rejects an unknown name. It proves that the lookup ignores case (tested for Python); that the code set is the code read back; and that the factory constructs and forwards. The smoke test it narrates says in its header: no `Execute`, by its own guardrail. `governance_narrated`, scene six, proves that an execution without a posture refuses and that a declared posture proceeds and names itself; scene seven, that postures travel through save and load. `virtual_environment_twin_narrated` proves that a queued spawn does not run until commit, by a marker file; that a scope refuses; and that the plan narrates and flags the spawn. `pycodegraph_narrated` and `jscodegraph_narrated` prove classes, methods, inheritance chains, owners, impact and cascade over Python and over JavaScript sources, offline and without the runtime. Both skip their parser scenarios when the engine's tree-sitter is absent, and the first also skips its Python-AST ones when Python is not on the path.

I want to be exact about the court of the door. A guard that runs a foreign runtime and passes is absent from the snapshot; what the door's court holds today is a registry and a construction test. The folder's own run record, under an older runtime version, lists thirty-seven failures. Several promises cannot match by construction, because the file names carry a random suffix and the calls a duration.

The field comes next, and where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in speed, and speed is what the door exists for. The article on performance measures seven runtimes on three tasks, and the host comes last in two of them. I carry its caveats, which chapter 22 states too: it has no versions and no dates inside its text, and it says that its times are relative to a specific computer environment. Sorting a million elements takes the host about thirty seconds and Python about four, in the article's table. I print no more of it.

The field leads in runtimes it can actually run. Of six doors, two ran on this machine, Node and Python; the R door started its runtime and raised; the other three had no runtime to start. The study says its description of the field's polyglot tools was written from the genre and not re-read, and I claim no measured comparison.

The field leads in the ecosystem and in being used. The compare page speaks of Python only as a rival; to cover its thirty rows in Python you assemble numpy, pandas and others. The door is how a Softanza solution takes them without leaving. I hold this as a view, not a result: section 3 shows the door carrying two runtimes in one pipeline, and nothing on this page runs numpy or pandas.

The court's reach stops today at the door: the foreign process is outside it, and chapter 21's threat model puts code already running in the process out of scope.

And the cost of a facet, in the memo's own table: a Python reader expects zero-based indexing, and Haro's is one-based. The literal `\n` is a backslash and an `n`; two letters of different case are one variable. "Any model writes Haro in the shape it knows" is true only of the spellings that are the minority.

7. THE LIMIT

BUILT	a closed registry of six foreign runtimes and a process-based call that brings Python's and NodeJS's results home as lists; the trust postures; the twin's rehearsal of a spawn; a reader of Python and of JavaScript into one code graph
DESIGNED	the door governed, with a posture asked at the call; the engine's bridge and its three tiers; the domain library on top of the door; Haro wearing a Python syntax and a JavaScript one
ABSENT	a door whose way back is closed; a door for R that runs; a passing run of a foreign runtime in the library's own court; the AI model as a door

The word *governed* is a design. Does the door ask the governance before it runs? This is the block that shows it:

HARO

```
oGov2 = new stzGovernance("shop")
? oGov2.MayExecute("python")
#--> 0
oPy3 = new stzExterCode(:python)
oPy3 {
        SetRuntimePath(cPython)
```

```
        SetCode("res = 6 * 7")
        Execute()
        ? Result()
        #--> 42
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The governance said no, and the door ran anyway, and returned `42`! Nothing in the door's folder mentions governance or posture. The door calls the system directly, and not the twin.

The way back is not yet closed. The result file is read by the host's own `eval`, and a string that comes out of the foreign runtime is wrapped in single quotes without escaping. Can a foreign string that carries a quote and a call reach host code? The function used here is `upper`, harmless on purpose:

HARO

```
oPy4 = new stzExterCode(:python)
oPy4.SetRuntimePath(cPython)
oPy4.SetCode(`
res = "x' + upper('y') + 'z"
`)
oPy4.Execute()
? oPy4.Result()
#--> xYz
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The result file held `'x' + upper('y') + 'z'`, and `Result()` evaluated it. The data became a call, and the block printed `xYz`! A plain apostrophe in data, `it's fine`, writes a string the host cannot read, and the method prints the error and returns the raw text. It is the same class of open limit as chapter 11's list in a natural sentence. I did not test the other five languages' marshalling.

The R door is in development: today it raises on every call, because its generated script holds an identifier that begins with an underscore, which R rejects. I confirmed it with R on a one-line file. And the trace records an exit code of 0 for a call that failed. The code passes a literal 0 to the recorder, and failure is detected only by the absence of the result file:

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
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The call raised, and the trace still holds `0`! R's message was in French on this machine, so I do not promise it. This is the limit of the door at this commit, in the voice of chapter 11's open list. The runs above are its evidence. Governing the door is designed and not yet built, so the vision's word "governed" stays a design. The way back is not yet closed, since data returns through a file that the host evaluates. And the R door, in development, raises on every call today.

What else is designed: the engine's own polyglot bridge, which the engine design marks as planned. A module of the same name in the macroplan, a string registry, was once marked done with no source. Then the three tiers for Python, of which

only the external subprocess exists. Then the domain library on top of the door, none of whose classes I found in code. And the facets, a memo that proposes and rules nothing.

What is absent: the article's functions that bring data in, `@@R`, `@@C`, `XC()` and `XAI()`, which are defined in no file. Factories exist for Python, R, Julia and Prolog. The C and NodeJS doors are reached only by the class. The AI model is not a door of this plane at all; the article's function that runs it inside the host is the opposite of chapter 17's law, and the book does not teach it.

And a door shape I did not run: Julia, C and Prolog, for which no runtime is installed here. And a third relation, named in the study beside the door: the shim that pastes a solution found on the internet in another language and changes it a little. `range0`, Python's range as a host function, has a guard. The folder's own record lists for it five failures on ranges that should be empty and one on the slice form `':5:-1'`. I ran none of it.

Six readings of the door follow, one for each person who meets it.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You write Haro and, for the layer another language does best, you hand it a string of code and a list of data and take a list back. In the plan, a Python or JavaScript programmer will also write Haro in a syntax they know. Two different things: a syntax brings you, and a runtime brings your libraries.
The analyst	The statistical layer is R's, the machine-learning layer Python's, the logic layer Prolog's. You stay in one file and one list of results. The R door is in development and raises today, and the record says so.
The designer	There is nothing to design at the door itself. The record's lesson is its refusal: a language not on the list is refused by name, and "what about my language" has one answer.
The decision-maker	The door is the one place where the closure ends, by decision. At this commit its governance is designed and not yet built, and its way back, not yet closed, reaches host code. The vision's word "governed" is a design. Every call's duration is recorded in the trace today.
The educator	A learner sees which layer belongs to which language and runs a two-door pipeline. The lesson is the difference between reading a language, running it, and writing in its syntax.
The agent	You never reach the door by yourself. A function in an AI agent's file must carry a posture, and `external` fits a foreign runtime; and an AI model has no capability to produce effects (chapter 20).

9. EXERCISE

You are building a small solution with three jobs: sorting a million numbers, deciding which of two students passes, and checking a family tree for who is an ancestor of whom. For each job, choose the language from the list of layers in section 3 that owns its layer, and write your reason in one line. Then, without running anything, write the promise for each line of this block, and run it:

```
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 ...
```

✎ not run: an exercise, written for you to run

The last promise is the one that matters. If you now call the door for Prolog, will the governance have been asked? Write yes or no, and why. Then read section 7. If a promise breaks, do not change it: explain the output first.

I left this one in your hands. Section 7 says that a plain apostrophe in data makes the way back write a string the host cannot read, and I printed no output for it. Here is the block of section 7 with one change, `it's fine` as the foreign string:

HARO

```
oPy4 = new stzExterCode(:python)
oPy4.SetRuntimePath(cPython)
oPy4.SetCode(`
res = "it's fine"
`)
oPy4.Execute()
? oPy4.Result()
```

✎ not run: an exercise, written for you to run

Run it. Read what the method prints and what it returns, and explain both before you touch the block; the page says why in section 7, where the way back is not closed. When you can, you will know why a way back that evaluates its data is not a closed one.

Chapter 17 opens the part on intelligence with its law: programmatic by default, the model last, admission over generation. An AI model there is one faculty among others, the last one called, and what it says is a proposal that the court admits or refuses.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	architecture
RULINGS OF THE STUDY	12.19, 12.20, 12.21, 12.22, 12.28, 12.29

PART IV

Intelligence, rethought

> *Programmatic by default, the model last, admission over generation.*

17 · Law: programmatic by default, the model last, admission over generation
18 · The five registers, and conversations that ask
19 · The domain language model
20 · Agents that cannot hurt you
21 · The court: promises, perception, security and performance
22 · Compared, honestly

PART IV · CHAPTER 17 · SPINE

Law: programmatic by default, the model last, admission over generation

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 8 of 8 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I state the law of Part IV and run it. I begin with my definition of intelligence and a domain language model with no neurons at all, which answers, says why, and refuses what it was never told. Then I show a conversation that asks you, a fact nobody recorded proved from one declared law, an AI model held to proposing, and an agent whose mind is code, refused at the gate. I say what I refused, which guards prove each block, and where the field leads. Then comes the limit, printed as the stage it is: a question asked in other words, answered where a refusal is the next step. The law itself is one sentence: the mind is programmatic by default, a model is one faculty among others, the last one called, and what it says is a proposal that the court admits or refuses.

WHAT THE AI AGE CHANGES HERE

In the two movements before this one, the AI age changed one thing each. In programming, it changed who writes the code. In languages, it changed who speaks the sentence. Here it changes everything, because this movement is about intelligence itself, and the field now mostly uses the word to mean one thing: a large language model inside your software.

So this chapter does what the other two law chapters do, and states the law of its part before the evidence. It also says how the part depends on the others, because the order of this book runs from the familiar to the new. Intelligence, in Softanza, is governed by what came before it. The court that judges a model's sentence is the court of Part II, which judges code whoever wrote it. The sentence it judges is a sentence of a closed language, which is Part III. A model can only be put last because there is something solid to put first.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Ask the field what makes an application intelligent and the answer is an AI model. An AI agent, in the field's sense, is a model in a loop with tools, a memory and a goal. The shopping list for building one in 2026 is long. It takes a project in Python, a model provider, a framework that wires the loop as a graph, a store of vectors for memory, tools with typed schemas, a tracing service, and a chapter of safety advice. Memory is often retrieval: a collection of documents, cut into pieces, searched by similarity and pasted into the prompt.

Safety, in that picture, is a property of the agent. It is alignment, guardrails, a prompt that asks for confirmation, a box marked "human in the loop". The field has also learned to hold a model's output to a shape, with grammars that constrain what it may emit. All of this is serious work, and some of it is good. It shares one assumption: the model is the mind, and everything else is around it.

2. THE FIRST PRINCIPLE

The mind is programmatic by default; a model is one faculty among others, the last one called, and what it says is a proposal that the court admits or refuses.

3. THE SOFTANZA FORM

Begin with my definition, because everything here follows from it. Intelligence is the computational ability to derive new knowledge from existing knowledge. Nothing in that sentence needs a model.

Here is a domain language model with no neurons at all, forged from a small knowledge graph of a restaurant. What does it answer from what was recorded, and what does it say about the 1998 world cup? Run it:

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).
}
```

✓ ran: printed what this block promises

It answers, says why (`recorded fact(s): 'margherita' contains tomato-sauce, mozzarella`), and states its certainty (`1`). And when it is asked something it does not know, it does not guess. It refuses, and it tells you how small its world is: `I know 1 entities and 1 relations`. One entity, because only `margherita` was declared with `Know`, and one relation, `contains`: the objects of the two facts are not counted as entities. A refusal that names its boundary is worth more than a fluent answer that has none. This is my view.

Now the second piece, the one that turns the usual dialogue around. In the field, you prompt the machine and keep what comes out. Here the system asks you. What does it ask, and what happens if you try to conclude before you have answered?

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.
}
```

✓ ran: printed what this block promises

A goal was declared: every dish must say what it contains. The gap between the goal and what is known generated the question, and the question carries its reason. The world cannot be closed while it is incomplete: `Can't conclude: 2 gap(s) remain`. Chapter 18 shows the five ways you may answer: an option from a list, a data structure, a formula, a sentence, or examples. Three are wired; the formula and the examples are absent. An AI model, where there is one, comes after all of them, not first.

The third piece is derivation, my definition at work. One law is declared, and a fact nobody recorded is proved:

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'
}
```

✓ ran: printed what this block promises

New knowledge from existing knowledge, with its chain written out! And a "no" that is as certain as the "yes", because it is the result of a search, not of a mood. The page printed it as `not derivable: no recorded fact or lawful chain gives 'margherita' contains 'luigi'`. It is certain about what is recorded and declared, not about the world: it says `not derivable`, not "false". No retrieval, no training. Adding a fact makes the system know more; adding a law makes it reason further.

Then the AI model. It is welcome, and it is held. Wire a model's output straight to an effect, here sending a message. Is the composition sound?

HARO

```
oG = new stzAgentGraph("mailer")
oG {
        AddLLMActor("writer")
        AddEffect("send")
        Proposes("writer", "send")
        ? IsSound()
        #--> 0
        ? len(Violations())
        #--> 3
}
```

✓ ran: printed what this block promises

It is not: `IsSound()` printed `0`, and it is refused before anything runs! Three laws fire, and `len(Violations())` printed `3`: every effect must pass a gate, open text must have somewhere safe to land, and every effect must leave a witness. What does a model actor hold that can act? No AI model is loaded in the block that asks: the answer is seeded, the way a test fixes a golden answer, so that the page runs the same on every night. Ask one:

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
}
```

✓ ran: printed what this block promises

Nothing. A model actor holds no effectful capability at all; it proposes, and that is the whole of its power. `HoldsEffectful()` printed `0`. What the block proves does not depend on the model: whatever it says, it holds nothing that can act.

And the programmatic agent itself has a mind made of code. A skill is a condition and an action, and it fires on what the agent perceives. Governance gates the act before it happens. When the stock is low and the skill wants to order, does the act go through?

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'
```

✓ ran: printed what this block promises

The stock is low, the skill wants to order, and the agent was never given the permission. `Pursue()` printed `0`: governance refused it! Nothing happens, and the log says why, by name: `refused: 'rogue-bot' lacks PERMISSION (can) for 'order-stock'`. The world is as it was.

Put the pieces in a line and you have the stack the vision of the platform names as its North Star, ratified on 27 September 2026: perceive, decide, react, learn. None of it needs a language model. It is a design, not an assembled agent. In it, perception classifies the world into the knowledge graph. A planner decides. Rules react. The agent scores its plans and keeps the best. The pieces exist in the library beside the agent, not yet inside it: the agent above calls none of them. A model may sit beside each step as one more faculty, and none of them depends on it.

Chapter 18 tells the five registers of an answer and the conversation that asks. Chapter 19 tells the domain language model, from its first rung, with no neurons, to its second, trained on the first one's own facts and nothing else. Chapter 20 tells the agent that cannot hurt you: it proposes, a workbench rehearses, the court judges, and only a governed actor commits. Chapter 21 tells the court itself. Chapter 22 compares all of it with the field, and prints where the field leads.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused the AI model as the mind. When the models arrived, the move was to put one at the centre and fence it. I wrote the opposite into the architecture: the mind is programmatic by default, and a model is a pluggable faculty, one among several, never the definition of the mind. The industry's loop is a special case of this design, with one faculty and no governance. The earliest sentence of mine the record holds on this is the architecture of July 2026.

I refused anything paid or remote as a requirement. Every capability has a floor that needs no setup, no subscription and no network. A model can be added on top. It is never the price of admission. "Last" means after the programmatic path, and under the court and a budget. Whether the last proposer can be a remote model under that budget, or must always be a local one, the record allows both readings, and I do not choose between them here.

I refused guessing. Every ambiguity refuses rather than guesses. The field's habit of prompting and keeping what comes out has a name now; here, guessing is replaced by asking, and vibes are replaced by governance.

I refused retrieval as memory. Memory is a governed graph, not a pile of similar fragments. Derivation replaces retrieval: the system knows what it was told and what follows from it by law, and it can prove both.

I refused a teacher I cannot see. The domain language model's second rung is trained on the facts of the first rung's own knowledge base and on nothing else. No remote AI model teaches it. The design goes further and says the second rung's training text is validated by the first rung's grammar before it is admitted, so that it is correct by construction. In the library the facts are rendered directly into text, with no such check yet; chapter 19 says where that stands. I gave the concept of a

domain language model, first designed for one of the estate's products, to the open foundation. Any project that supplies a knowledge base can ship its own model free.

I refused a stub. The library declares five agents of its own and has wired one. Asking for one that is not wired raises an error; it does not return something that looks like an agent. Pretending would be easier, and it would make every other refusal in this chapter worth less.

Behind the blocks stand six guards, and the next section names them.

5. THE COURT

Each block above stands on a guard. `dlm_narrated` proves the domain language model: answers with their why, refusal outside the domain, completion held to the domain's entities and relations, a self-contained bundle, and a second rung trained on its own corpus. `wisecoding_narrated` proves the conversation that asks and the refusal to conclude. `knowledge_integration_narrated` proves derivation and the certain "no". `agentgraph_narrated` proves that a model straight to an effect is not sound, that the governed shape is, and that a model actor proposes and never commits. `piagent_narrated` proves the programmatic agent and its refusal at the gate. `ownagentstack_narrated` proves the honest roster of the library's own agents.

These six guards prove what ran; the next section says where the field leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in reach. Its frameworks connect to hosted services of every kind, its stores search open collections of documents, and its loops call the largest models available for each step. Softanza's inversion, curate a domain and derive from it, leads on a curated domain, and on the open web the field leads today. When the task is open-ended language work over material nobody can curate, summarising arbitrary documents or research over the web, the field's stack is the right tool. The record says so in those words.

The field leads in the market. There is a labour market for its stack today; the one for this stack is still to come.

The field leads in AI models. The second rung of the domain language model, as built, is a small neural model of word pairs, and the foundry beyond the bigram is designed (chapter 19). Softanza does not build the large generators and does not compete with them.

Where I think the law stands apart is in three places. A decision here costs nothing, runs offline and gives the same answer every day, so it can be tested with ordinary promises. An AI model here cannot act, by construction, not by vigilance. And a model's sentence, where one is used, can be held to the grammar of a declared language before it is emitted, instead of being checked and retried after.

The next section says where the law stops.

7. THE LIMIT

BUILT	the domain language model's first rung and its second, a small bigram; the conversation that asks; derivation with its proof; the programmatic agent; the gate that refuses an ungoverned AI model
DESIGNED	the thinking machine's cascade across faculties; the decision stack, its pieces beside the agent and not yet inside it; the typed unit of prediction, a design of the estate's with five validation stages and no chain of them in the library; the negotiation between an agent and its reviewer
ABSENT	four of the library's five own agents, induction from examples among them

What is built: the domain language model's first rung and its second, a small bigram, the conversation that asks, derivation with its proof, the programmatic agent, and the gate that refuses an ungoverned composition.

What is designed: the thinking machine of chapter 9, where many faculties share one state and cascade until it settles. The decision stack of perceive, decide, react and learn, whose pieces exist beside the agent and not yet inside it. The typed unit of prediction, which chapter 19 tells as a design of the estate's, validated in five

stages in that design and chained nowhere in the library. The conversation between an agent and the one who reviews its plan: the plan now reaches the agent, and the conversation about the plan does not yet.

What is absent: four of the library's five own agents. How many does the library declare, and how many are wired? The roster says it plainly:

HARO

```
? len(StzOwnAgents())
#--> 5
? StzOwnAgentIsWired("wise-coder")
#--> 1
? StzOwnAgentIsWired("induction")
#--> 0
```

✓ ran: printed what this block promises

The validator, the ranker, the planner and the agent that would induce a pattern from your examples are declared and reserved. So the fifth register of an answer, examples, waits on an agent that does not exist yet.

And one limit of the first rung that you will meet the moment you rephrase a question. It recognises a relation only by the word it was declared with. What does it do when you ask in other words?

HARO

```
? oDLM.Ask("what does margherita contain?")
#--> Margherita is a dish.
```

✓ ran: printed what this block promises

It did not refuse! It printed `Margherita is a dish.` That answer is true, and it is not what was asked. By the law of this chapter, the answer here is a refusal, and that refusal is the next step. I print it here because a law is shown by a page that prints its stage as it stands, limit included. The request to the library is to refuse a question whose verb is not a declared relation.

8. SIX DOORS

Now the exercise: your own domain, and the promises you expect.

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your agent's mind is code you can test with ordinary promises. An AI model may serve as one faculty; it can never hold an effect.
The analyst	A fact or a rule you add makes the system know more, with no code change and no training, and every answer carries its why.
The designer	The system asks the user one accountable question at a time, and the question is handed to any door as a frame, so that a terminal, a page or an agent can render it. Only the terminal ran here.
The decision-maker	A decision costs nothing per call, runs offline and is the same tomorrow. The four agents still reserved are the next step.
The educator	The conversation asks and never answers first, and it uses a model's fluency only when a model is there, so a learner always sees the floor.
The agent	You may propose. You may not commit. A refusal names its law, and you can learn the platform from the platform and be corrected by it.

9. EXERCISE

Build a domain of your own with three facts: a shop, a clinic, a class. Before you run anything, write under each line the promise you expect: one question inside the domain, one outside it. Add a goal that every item must have one relation, and write the first question the system will ask you. Then run it.

If a promise is broken, do not change the promise to match the output. Explain the output first.

I left this one in your hands, on purpose. Section 7 shows a question that the first rung answered when the law says it should have refused, and I did not fix it. Take your own domain and ask one of your promised questions again in other words, with a verb you did not declare as a relation. Look at what comes back and compare it with what you asked. Then say in one sentence why it answered, and which lines of section 7 tell you so.

When you have solved it, you will know why a first rung that recognises a relation only by the word it was declared with can answer a question you did not ask.

Chapter 18 is *The five registers, and conversations that ask*. Its principle: a reply may come in any of five forms, and whatever the form, it is only a candidate, and the world's rules decide whether it becomes knowledge.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-agents-that-cannot-hurt-you-narration.md`
NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stz-knowledge-programming-narration.md`
ON THE SITE	agentic
ON THE SITE	wise
RULINGS OF THE STUDY	12.18, 12.19, 12.25, 12.26, 12.29

PART IV · CHAPTER 18

The five registers, and conversations that ask

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 15 of 15 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I show you the five forms a reply can take, an option, a data structure, a formula, a sentence and examples. Then I show what the governed door does with each of them. I tell you which three run and which two do not, and what becomes of a refusal once it is written: it waits for a person, with its context and a lifetime. I let the conversation with no one in it run, and I show what a conversation can and cannot do in Arabic and in Hausa. Whatever the form, a reply is only a candidate, and the world's rules decide whether it becomes knowledge.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Two practices meet in the idea of a conversation that asks, and a third stands behind its last register. This whole section is general knowledge, not a study of the record.

The first is slot filling. A dialogue framework asks the user for the missing pieces of a form until the form is full. The frameworks of the field understand free text with trained models, speak on many channels and ship connectors, and they ask the designer to write intents and flows in their own language. They take the user's reply as the answer and act on it.

The second is vibe coding, a name put into circulation in 2025: the human prompts, the AI model produces, the human keeps what runs.

The third is programming by example. Spreadsheets fill a column from two examples you typed. Research calls it inductive synthesis, and the model era made the example the way one steers a model.

2. THE FIRST PRINCIPLE

A reply may come in any of five forms. Whatever the form, it is only a candidate, and the world's rules decide whether it becomes knowledge.

The design document says it in one sentence I keep: expression is free, admission is governed. The record holds this sentence, captured from my directive of 13 July 2026. It holds no sentence of mine on why these five registers and not four or six, nor on the moment I set wise coding against vibe coding.

3. THE SOFTANZA FORM

Chapter 17 showed a conversation asking, and refusing to conclude while a gap is open. This chapter is about the answer. The five registers are five forms a reply can take: an option picked by its number, a data structure, a formula, a sentence, and examples. I will show you each, and I will tell you the ones that do not run.

The first register: an option. When the world already knows values, how does the question offer them? Run it:

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" ]
}
```

✓ ran: printed what this block promises

The question offers the known values as numbers, and `2` picked `olives`.

What does the door answer to a number nobody offered? Run it:

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
}
```

✓ ran: printed what this block promises

The door refused it, and said `no such option`! The checkpoint says why: `no option (99) was offered -- 2 option(s) on the table`. A reason, and a count of the options on the table. It does not list them, and the next question offers the same options again: `(1) cheese  (2) olives`. The refusal never guesses a value, it reaches a person as a checkpoint, and the goal stays `pursuing`. That matters for section 7.

The question is data before it is text. It carries a force, which is *what* while the world knows nothing and *which* once it knows a value, and the options. Any medium can render it:

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" ]
}
```

✓ ran: printed what this block promises

The second register: a data structure. What becomes of a list of two values sent as one reply? Run it:

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
}
```

✓ ran: printed what this block promises

A list of values is taken as the values, and each one goes through the same door.

The fourth register: a sentence. I skip the third for a moment, because it is one of the two that do not run. How is a phrase of the form "X and Y" or "X, Y" read? The conversation keeps a transcript in which every admission is narrated

and the fulfilment of the goal is recorded:

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'
}
```

✓ ran: printed what this block promises

It is read as two values, each one judged: the transcript holds `admitted: 'margherita' contains 'tomato-sauce'` and `admitted: 'margherita' contains 'mozzarella'`. "Natural" here means a splitter on commas and on the English word "and". The natural layer of the library, which I will use in a moment, is not called by the conversation.

When a reply carries several values, and the world allows one. What does a relation declared unique do with a reply that carries two values? Run it:

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
}
```

✓ ran: printed what this block promises

The first value was admitted, and the second was refused with the law's reason: `'kitchen3' already bears 'led-by' (to 'mario'); the relation is :Unique`. The refusal reaches a person, as one checkpoint. Be careful what you read in this. The choice between Mario and Giovanni was made by order. It was not made by a person. The design promises something else, and section 7 says what.

The third register, the formula. Is a formula typed as a reply evaluated? Run it:

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
}
```

✓ ran: printed what this block promises

It is not. The price of the tea is now the text `=12*3`! I found this out by running it, and the code is silent about a formula register. The record lists the wise-coding work on registers 3 and 5 as waiting, not routed.

The fifth register, examples. What does the door do with examples offered as pairs? The agent that would read them is on the roster, and reserved:

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().
```

✓ ran: printed what this block promises

Neither admitted nor refused: `[ ]` and `[ ]`, the goal still `pursuing`, and not one checkpoint! This is not the refusal the protocol describes. It is silence. I hold silence to be the worse answer, as a view and not a result: the reader who offered the examples is not told they were not understood. I found a second form of the same fault, with the same examples typed as one string, joined by a comma. The exercise at the end of this chapter hands it to you.

So three of the five registers are wired, one of them in its simplest form, a splitter. Two are not. I would rather you meet that here than on a page of mine that said five.

That leaves the refusal itself, and what becomes of it.

WHAT HAPPENS TO A REFUSAL

A refusal waits for a person, with its context. A lifetime can be set so that a stale one leaves the live list without being deleted:

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
}
```

✓ ran: printed what this block promises

Nothing is deleted. Expiry is about what still needs a person, and the audit record keeps all of it. Every call here is chained from `ConversationQ`, because a held copy would silently do nothing: the guard's comment explains it.

A conversation is accountable for one goal, and it ends by fulfilling the goal or by revoking it on the record. A revoked conversation asks nothing and cannot conclude:

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
}
```

✓ ran: printed what this block promises

PROGRAMMATIC BY DEFAULT: THE CONVERSATION WITH NO ONE IN IT

Chapter 17's law says conversations happen in code first. Here is the conversation with no person and no model. An AI agent of the library asks, an answer source (a plain function) replies, and every reply passes the same door. Where the source has no answer, does the agent invent? Run it:

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
```

✓ ran: printed what this block promises

It does not. It stops, says `stopped: no answer for 'panna-cotta' / 'contains' -- gap left for a human`, and leaves the gap for a person. Of the library's five own agents, this is the one that is wired. The other four are reserved, and asking for one raises, as the fifth register's block printed: `RESERVED, not wired yet -- refusing rather than returning a stub`. It is never a stub.

IN ARABIC, AND IN HAUSA

Chapters 2 and 11 showed the natural layer in Hausa. Here is what the conversation can and cannot do with languages, said in three blocks, because the three are different.

First, the data. Names, values and lists in Arabic script pass the gate exactly as Latin ones do. The words around them, the frame of the question, stay English:

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
}
```

✓ ran: printed what this block promises

The Arabic values passed the gate, and the goal reads `fulfilled`.

Second, the candidate. A Hausa sentence can produce a value in the natural layer, and the value, sent as a reply, goes through the same door. The two steps are the caller's: the conversation does not call the natural layer itself. This is the principle of the chapter in a small space, since the expression is free and the admission is governed. The graph folds the case of the value, so the printed `JIBINI` is admitted as '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'
```

✓ ran: printed what this block promises

Third, the question in the learner's language. The tutor is a consumer of the conversation: it uses the gap the conversation names, and it chooses the words to say it in four languages. The sentences are templates. The code's own comment says that the French, Arabic and Hausa wording is a draft awaiting a native reviewer. Read the three below as drafts, and not as reviewed text:

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)
}
```

✓ ran: printed what this block promises

So the library cannot ask in Arabic or in Hausa through `AskIn`. What it can do is hold the data in any script, take a candidate computed in any pack, and let the tutor say the gap in four languages. A value with a space in it raises at the moment (section 7 gives the evidence). And the splitter knows the comma and the English word "and" only: the connectors "et" in French, "da" in Hausa and "و" in Arabic are not connectors. The three sentences above are the library's own, printed as they stand, and the code calls them a draft awaiting a native reviewer. I wrote no Arabic or Hausa sentence of my own.

AN AI MODEL IS A FLUENCY, NEVER THE MIND

Without a model, what does a conversation say of its own fluency? Run it:

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
}
```

✓ ran: printed what this block promises

`plain`: it stays on its deterministic floor and says so. The word "neural" can be set, and `Fluency()` repeats it, but `IsFluencyNeural()` answers `0`! The word is claimed only when a model is there.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused to guess. The record's phrase is: guessing is replaced by asking; vibes are replaced by governance. A gap is a question. A reply is a candidate.

I refused a different gate for each way of answering. A formula is not trusted because it is code, and a structure is not trusted because it is typed. The form does not buy trust. All five registers go through one door.

I refused a stub. A reserved agent raises when it is asked for. The design document calls the raise the law working, and says it should not be read as a defect. I refused to pretend the fifth register works.

I refused an AI agent that fills a gap to finish the job. The wise coder stops, and leaves the gap for a person.

I refused to let a conversation own a world. It happens inside a knowledge space and never owns one. A copy of the knowledge held in a session goes stale.

I refused a deleted checkpoint. Expiry takes a refusal off the live list. The record keeps it.

I refused to let the example look like the answer. The record's closing law about the fifth register is that an admitted candidate writes its example back as a promise. That law is designed, and not built.

5. THE COURT

`wisecoding_narrated` proves that the gap is computed from the space, and that the question is led by the system, names its subject and says why it asks. It proves that "X and Y" and a data structure admit. It proves that a unique law refuses through the conversation and the refusal reaches a checkpoint. It proves that

concluding with a gap open is refused. `wisecoding_rich_narrated` proves the goal monitored to fulfilled or revoked, the frame, and the numbered options. It proves the refusal of a number nobody offered, the lifetime of a checkpoint, and the fluency that is claimed only with a model. `ownagentstack_narrated` proves the roster of five, the wired wise coder, the reserved agent that raises, and the gap left for a person. `tutor_gaps_narrated` proves the exercise's gap question in each language, and that a step with no question in a language is red, never another language. It writes folders, so the book counts it and does not run it; the tutor block above only reads the course.

Three of the guards print their own total and end by calling `pf()`, which prints the time and the runtime's version and then stops the run with a raise. That is their way to end. `tutor_gaps_narrated` ends with `Summary()`, which prints the total and a status, and does not raise.

What the field does better, and where the record stands apart, comes next.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in understanding. A dialogue framework, as the field holds it, reads free text with trained models, speaks on many channels and ships connectors. The record says what Softanza lacks, in its own count: the word "WhatsApp" appears in no code file and no design document of the library. The conversation asks one slot at a time, over a graph the user must first have declared, and it does not understand an open reply.

The field leads on examples. Programming by example is, as the field holds it, shipped in spreadsheets today. Softanza has no induction at all, and the record says so: nothing induces yet. A search for "induce", "from examples" and "infer" over the regular-expression folder finds nothing.

The field is honest about costs, and so is the record. Several candidates is the normal case when you ask for a rule from examples. A pattern induced from three examples overfits them. The remedy the record names is the question that tells two candidates apart. And the index that would carry the examples is, in the study's words, as clean as the tags of the methods, in which the study counts about eighty-five misspellings.

Here is where I think the record stands apart. Every reply, in whatever form, is judged by the world's laws before it counts, with the reason. The session keeps a transcript and a checkpoint that a person can read. The field's reply is a prompt, and the human keeps what comes out.

The limit comes next, and it names what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	the goal, the gap and the question with its reason; a reply as an option, a data structure or a sentence split on commas and the word 'and'; the governed door with its refusals; the checkpoint with its lifetime; the goal fulfilled or revoked; one wired AI agent that stops where it would have to invent

DESIGNED — several readings of one reply, enumerated for the person to choose; the nearest alternatives named after a refusal; the example written back as a promise

ABSENT — the formula register; the example register; a conversation whose question is worded in Arabic, Hausa or French; a value with a space in it

What is built: the goal, the gap, the question with its reason, the frame; the first register, the second, and the fourth in the form of a splitter; the governed door with its refusals. And the checkpoint and its lifetime; the goal fulfilled or revoked; the transcript; one wired agent; the tutor's four-language gap sentence.

What is designed: the *several acceptable* branch as the design writes it. The protocol says that when several readings of a reply are acceptable, the system enumerates them and the person chooses. What is built is two different things. The values already known become numbered options in the question, before the reply. And a reply that carries several values has each one judged on its own, with the first kept under a unique law. No step presents competing readings of one reply. And the *none acceptable* branch as written: the reason is built, and the nearest acceptable alternatives are not. The refusal states a reason, and for a number, how many options were on the table. And the example written back as a promise: the site's page *by example*, read on 9 October 2026, calls it a rule of the design that is not built. The site's two pages on this, *wise* and *by example*, say "with the nearest alternatives" and "the person chooses". The two pages promise more than the code does. The code wires three of the five registers. The record names the formula and the examples as waiting, not routed, and for this chapter they are absent.

What is absent: the formula register; the example register, and the induction agent behind it; a conversation worded in Arabic, Hausa or French; and a value with a space in it. In my run of 8 October 2026, on the version of the library this book runs, `ReplyIn` raised on each value containing a space that I tried. The error was an uninitialised variable. The three were "tomato sauce", "fromage et tomate", "jibini da tumatir". Called directly, the door returned the refusal properly. No

block of this book records either result, so the run of the book does not show them. The conversation should have refused with the reason and written a checkpoint, and it doesn't. It is a request to the library, and the exercise below does not use a value with a space.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	A reply is a verdict you can test with ordinary promises: what was admitted, what was refused, and the state of the goal.
The analyst	The goal is the schema of a complete model. The gaps are the list of what is missing. The transcript and the checkpoints are the evidence of how each fact got in.
The designer	The question comes back as data: force, subject, relation, options, reason. Any medium can render it. The library renders English, and the tutor shows the pattern for four languages.
The decision-maker	A reply is judged before it counts. A refusal reaches a person with its context and a lifetime. The roster is honest: one agent of five is wired, and two of the five registers are not available.
The educator	The tutor is wise coding. It names the missing step as a question and never as the answer, in the learner's language, and its wording in three of those languages awaits a native reviewer.
The agent	You can drive the loop yourself with an answer source, and you must stop where you would have to invent. You cannot offer examples and be understood, and you are told by name when you ask for a reserved agent.

9. EXERCISE

Choose a small world of your own, a shop, a class, a clinic, with four entities of one type and a goal that each needs one relation. Before you run anything, write under each of the five replies below the verdict you expect: admitted, refused with a reason, or neither.

1. a plain word;

2. a number that picks a value the first reply made known;
3. a number nobody offered;
4. two values at once, for a relation you have declared unique;
5. a list of two pairs of examples.

Then run it. An admitted reply uses up the question, so call `AskIn` again before the next reply. If a promise breaks, do not change the promise: explain the output first.

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
```

✎ not run: an exercise, written for you to run

The fifth reply produces silence today, and that is the lesson of the exercise. If the library later turns it into a refusal, the expected answer changes. Run it on the date and commit shown at the foot of the page.

I left this one in your hands. Take the fifth-register block of section 3 and change one thing: send the two phone numbers as one string, joined by a comma, and not as two pairs.

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]
```

✎ not run: an exercise, written for you to run

Run it, then read what was admitted and what the goal says, and decide whether that is what you wanted to hand to the door as examples. Section 4, in its last refusal, and section 7, in the absent example register, say why I call it a fault. When you can say why, you will know what the door did with your examples, and why that is not induction.

This is what I call wise coding. The platform asks the question you have to answer, in the register you choose, and refuses to guess the answer for you: expression is free, admission is governed. Chapter 17 stated the law. This chapter ran its five registers, and showed where the fifth still falls silent.

The next chapter is about the domain language model. There the next thing the model says is a checkable unit of your world, a fact your rules admit or refuse, and never a token.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-knowledge-programming-narration.md`
NARRATION	`stznatural-narration.md`
ON THE SITE	wise
ON THE SITE	byexample
RULINGS OF THE STUDY	12.25, 12.26, 12.27

PART IV · CHAPTER 19

The domain language model

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 7 of 7 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I forge a domain language model from a knowledge base, and no one writes its training text: every sentence is a recorded fact. It has two rungs. The first answers from those facts and has no neurons. The second, a small neural model, is fluent in the grammar of the domain and wrong about facts. I show the grammar that refuses a malformed sentence and the graph that refuses a false one. I say which pieces the library holds and which the reader still has to compose. The principle is this: the next thing the model says is a checkable unit of your world, a fact your rules admit or refuse, never a token.

1. THE CONCEPT, AS THE FIELD HOLDS IT

A language model predicts the next token. A token is a fragment of text with no type, no role and no constraint: it can be the end of a word, a space, a bracket. Everything a model says is built by choosing tokens one after another. Everything that makes the result useful, that it is a valid sentence, that it is true, that it fits a schema, is checked afterwards or enforced from outside.

The field has made this work with three tools. You adapt a general AI model to a domain by training it further on the domain's documents. You give it the relevant documents at the moment of the question, retrieved by similarity. And you constrain its decoding with a grammar, so that it cannot produce a malformed sentence, as chapter 11 showed. All three keep the unit the same: the token.

The next section names the one thing I change: the unit.

2. THE FIRST PRINCIPLE

The next thing the model says is a checkable unit of your world, a fact your rules admit or refuse, never a token.

In the design the gate is the same for every model of this kind: generations are proposals, and the graph and its rules decide admission.

Section 3 shows the form it takes, and runs it.

3. THE SOFTANZA FORM

I call it a domain language model, and it has two rungs. The first has no neurons at all and was shown in chapter 17: it answers from recorded facts, says why, and refuses outside its domain. The second is a small neural model, optional, which gives it fluency. In the design both rungs end at the same gate; here the gate is the `Admit` of the last block of this section, called by hand.

Where does the training text come from? Build a small knowledge base, render it, and read the corpus and the vocabulary it gives.

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 ]
}
```

✓ ran: printed what this block promises

No one wrote that corpus: each of its six sentences is a fact of the knowledge base, rendered. The vocabulary is closed, every word the domain holds with a first entry for the unknown. Twelve entries, and `something-new`, a word outside the domain, maps to the first one: `[ 2, 7, 1 ]`. "Correct by construction" holds here in the one sense the code gives it: each sentence is a recorded fact. It does not yet hold in the sense the design intended; section 7 says where.

Completion is constrained to the domain. What does the model offer after an entity that bears facts, after a relation, after a fragment, and after a word the domain does not hold? Run it:

HARO

```
oDLM {
        ? @@( Complete("margherita") )
        #--> [ "contains", "pairs-with" ]
        ? @@( Complete("contains") )
        #--> [ "tomato-sauce", "mozzarella" ]
        ? @@( Complete("tira") )
        #--> [ "tiramisu" ]
        ? @@( Complete("pizza") )
        #--> [ ]
}
```

✓ ran: printed what this block promises

After an entity that bears facts, its relations: `[ "contains", "pairs-with" ]`. After a relation, only its objects: `[ "tomato-sauce", "mozzarella" ]`. A fragment completes to entities, and a word that the domain does not hold completes to nothing: `[ ]`. The guard that proves this calls it "the grammar spirit". It is a lookup in the facts, not decoding under a compiled grammar. And it sees one word, not the sentence so far: `Complete("contains")` returns the objects of every `contains` fact, whatever its subject.

The grammar itself can be derived from the knowledge base, and here the reader of this book has to write the derivation. There is no call in the library that turns a knowledge base into a grammar, and the design lists it as the first rung's task. But the pieces compose. The domain's vocabulary becomes the choices of a schema from chapter 11, the schema compiles to a grammar, and the grammar has no path for a word outside the domain. The three rules it prints last, for text, number and boolean, are the ones the compiler always emits, and no field here uses them:

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'.
```

✓ ran: printed what this block promises

Look at the last four lines together. The grammar accepts the lie! It has the right shape, and every word is in the domain. Asked what a tiramisu contains, the model answers: `I hold no 'contains' knowledge about 'tiramisu'.` A grammar is a gate for shape; the truth belongs to the graph.

A domain language model is a file that travels. It is saved as plain text, readable and diff-able, with no binary and no weights. Can a second model that has never seen the knowledge base answer from it, and run its own tests? Save it, read it back, and load it:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The second model answers: `Tiramisu is a dish.` The refusal travels with the bundle as a golden test, and the model that has never seen the graph passes it too! The laws section is empty because no law was declared before the model was forged; declare that a relation is unique, forge again, and the line appears.

Now the second rung. It is a small neural model trained on the corpus you saw, with no teacher, and it can be exported as a model file. How small is it, and does it learn?

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)
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

It is small: a single layer that predicts the next word from the previous one, `15 pairs over a 12-word vocab`. And it learns: `loss 2.57 -> 0.70`. The library inspects the exported file and prints `2`, the two tensors it holds, a weight and a

bias. I do not claim another runtime loads it as a text generator. It trains in a fraction of a second, offline, and gives the same numbers on every run.

What does a neural model like this say? It is fluent in the grammar of the domain, and it is wrong about facts. That is the reason for the gate:

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
```

✓ ran: printed what this block promises

"Tiramisu is a wine" is false, because tiramisu is a dish. "Chianti is a wine" is true. And "margherita contains mozzarella is" is a run-on, because the bigram has no word that ends a sentence. The test against the corpus is a crude gate, and it would refuse a true fact nobody recorded. The real gate is the graph and its laws. What does the graph answer when the false sentence is proposed to it as a fact? Declare the law, then propose:

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'
}
```

✓ ran: printed what this block promises

The graph refused the false proposal! It says why: `'tiramisu' already bears 'is-a' (to 'dish'); the relation is :Unique`. The refusal is kept, with its source: the model's second rung. A new fact is admitted. That is the whole of the principle in six calls: a proposal, a law, a refusal with its why, and the refusal on the record.

The design describes the unit of prediction in three levels. A token is a fragment with no type. An embedding is a vector without a role. And a chunk is a typed node of a graph, with a tag, a payload and edges: a declaration, a relationship, a constraint. A chunk is validated in five stages before it enters the graph: type, pattern, graph integrity, compliance, and constructive contribution. I tell it as design, because there is no chunk in the library. What exists are pieces: the court of chapter 11 checks a text's type and shape; and the admission above enforces a law, which is compliance. Nothing chains them.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

On 13 July 2026, I ruled on this. The concept of a domain language model had been designed for one of the estate's products. I donated it to the open foundation, so that any Softanza project that supplies a knowledge base can ship its own model free to the users of its domain. The format is `.zdlm`. The first rung is in every model; the second is optional fluency. The record holds the ruling that a model is free to ship, and no licence text. I tell "free" as the ruling's word: a plain file, in the open foundation, with no licence fee or meter.

I refused the token as the unit. The unit is a chunk of the world, which a graph can admit or refuse.

I refused a teacher I cannot see. The earlier design of the idea used a large commercial AI model to write its training corpus. I put a law against it, and this model's corpus is rendered from the domain's own facts: a law of the library says a capability never requires anything paid or remote.

I refused neural first. The finding I start from is that the canonical model is not neural. A review of the earlier product, recorded in the estate's design of it, archived a transformer of one to three billion parameters in favour of rule-based prediction. Its ground was that hallucination is impossible when the model cannot generate almost-valid text. So the first rung has zero training and zero cost, and the neural rung comes second.

I refused to let the shape stand for the truth. The grammar says structure kills malformedness, not falsehood. The truth belongs to the graph, to its laws, and to a person.

Section 5 says what the court proves of this.

5. THE COURT

`dlm_narrated` proves the model forged from a graph, its answers with their why, its refusal outside the domain and the completion. It proves also the plain-text bundle with its goldens, the closed tokenizer and the second rung with its model file. One of its assertions on the second rung checks only that the generated sentence begins with a fixed prefix, so it would also pass the false sentence above. `structuredoutput_narrated`, `schema_grammar_narrated` and `constrained_decoding_narrated` prove the court of chapter 11, the grammar compiler and the sampler. The last of them has two scenes that need a generative model file, which the snapshot does not hold; the record counts 50 assertions in that guard. `knowledge_integration_narrated` proves derivation and the refusal by a law. The admission I show in the last block is not called by its name in any guard. It is reached through the conversation, and by the strict form of adding a fact, in which a fact added without its admission is refused. That much the guard does assert.

Section 6 asks where the field leads, and where this model stands.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in scale. Training and serving AI models of every size is a mature practice. This second rung is a single layer over twelve words, and the steps that would make it more, a validated corpus, deduplication, a trained tokenizer, a larger network, quantisation, are not done. I do not compete with large AI models.

The field leads in fluency, which is the point of an AI model. A first rung that answers from facts will say "I hold no 'contains' knowledge about 'tiramisu'." where a general AI model would write something plausible, and for a customer the plausible thing is often more useful. The record's own answer is the second rung, and its present state is the one section 7 gives.

Where I think this stands apart is in what the output is. A general AI model's answer is text, and every check on it is after the fact. Here every answer of the first rung carries its why and a certainty of one. Every proposal of the second can be put

to a gate, and the gate can refuse it with a reason. A model that can only say what the domain holds, and a gate that remembers what it refused: that is what a bank, a ministry or a school can sign. I hold that as a view, not a result.

Section 7 says what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	the domain model's first rung with no neurons; its facts rendered as a corpus; a closed vocabulary; completion by lookup; a plain-text bundle that travels with its tests; a small second rung, a bigram, exported to a model file; the admission of a fact by a law
DESIGNED	a grammar derived from a knowledge base by one call; the typed unit of prediction validated in five stages; a second rung whose corpus is checked before it trains; the foundry that would make it larger
ABSENT	a pipeline that chains the five stages on any unit; an end-of-sentence word for the second rung; a completion that sees the sentence so far

What is built: the first rung and its answers; the facts rendered as a corpus, with a closed vocabulary; completion by lookup; and the bundle that travels with its tests. Also built: a second rung that trains in a fraction of a second and exports a small model file, and the admission of a fact by a law.

What is designed comes in four points.

- The grammar derived from a knowledge base by one call. This chapter's composition is the reader's, not the library's.
- The second rung's corpus validated by the first rung's grammar before it is admitted. The design says so; the code renders the facts directly, with no such check. So "correct by construction" holds for the training text, and not for what the model says.
- The five-stage validation of a chunk. None exists as a pipeline.
- The foundry's steps beyond the bigram.

What is absent: an end-of-sentence word for the second rung, which is why one of its sentences runs on. Absent too is a completion that sees the sentence so far; that is why a completion after a relation lists the objects of every fact of that relation. The first rung also answers a rephrased question with the type, as chapter 17 showed, where the law of this part says it should refuse.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	A domain language model is a file you can read, diff and test: its facts, its laws and its goldens.
The analyst	A fact you add makes the model know more, and a law you add makes the graph refuse more, with no training; the model's own answers do not read it yet.
The designer	The model's refusal is part of its contract: the golden test says what it must decline to answer.
The decision-maker	The model can only say what the domain holds, and the gate keeps the record of what it refused and why.
The educator	Rung one is a model a learner can read in full. Rung two is small enough to train in front of a class.
The agent	What you propose is a fact the graph can admit or refuse, with a reason you can read.

9. EXERCISE

Forge a domain of your own with three entities and two relations. Before you run anything, write what you expect from three calls. They are the completion of an entity, the completion of a relation, and the answer to a question about a thing the domain does not hold. Then declare that one relation is unique and propose, in the form of a fact, something that contradicts it.

Run it. Then write a sentence of the right shape and the wrong truth, and say which of the grammar and the graph refuses it, and which does not.

I left this one in your hands. Section 3 says that completion sees one word and not the sentence so far, and section 7 lists it as absent. In your domain, give two entities the same relation with different objects, ask for the completion of that relation, and read the list against the sentence you had in mind. The library does not repair it, and I have not repaired it for you. Then compose, from what section 3 shows (the facts of a knowledge base, read one by one), a completion that sees the subject as well as the relation.

When you have solved it, you will know why a completion that sees one word lists the objects of every fact of that relation.

Chapter 20 turns from what an AI model says to what an AI agent does. Its title is *Agents that cannot hurt you*, and its principle is this: do not build safe agents; build a safe world, and let ordinary agents loose inside it.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stz-knowledge-programming-narration.md`
ON THE SITE	languages
ON THE SITE	agentic
RULINGS OF THE STUDY	12.19, 12.29, 12.30

PART IV · CHAPTER 20

Agents that cannot hurt you

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the action half of what chapter 17 said of the whole mind: what happens when the proposal meets the world. I show four steps in code: the AI agent proposes, the workbench rehearses, the court judges, and only a governed actor commits. Then come the cases where a step does not hold. I say what I refused: safe agents, a permission check as the boundary, a second workbench for machines, an optional gate. And I say where it stops: a plan with no executor at all is allowed, and the loop checks that a coverage statement exists, not that it is true. The principle is the doctrine's: do not build safe agents; build a safe world, and let ordinary agents loose inside it.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In the field, safety is a property of the AI agent. The doctrine document of the library lists what the field offers: alignment, guardrails, confirmation prompts, a checkbox that says a human is in the loop. Tool-using agents, and the autonomous agents that descend from them, let the model's tool calls touch reality immediately, with a prompt for confirmation. Computer-use agents run in a sandboxed virtual machine, which contains a process and, in the doctrine's words, does not know a file move from random bytes churning.

The best advice of the field, which the narration says every serious guide of 2026 gives (it paraphrases one roadmap), is to separate planning from execution and put a human approval between them. It is good advice, and it is advice: a pattern

you implement, and can forget. The threat lists of the field, the OWASP list of risks for agentic applications among them, are lenses, and the record uses them so.

So the field's posture, in a line, is that the agent is an AI model in a loop, and safety is vigilance around it. That is general knowledge, not a study of the record, except where I name the document. Chapter 17 said this of the whole mind. This chapter takes the action half: what happens when the proposal meets the world.

2. THE FIRST PRINCIPLE

Do not build safe agents. Build a safe world, and let ordinary agents loose inside it.

The sentence is in the library's doctrine document: "We will not build safe agents. We will build a safe world, and let ordinary agents loose inside it." The document gives its structural half in one more sentence. The only artefact that ever leaves its world is an update plan, and plans are executed by someone else. The document carries no date and no author line in its header, and git first records it in a commit of 9 July 2026. The chapter holds the sentence as the document's, dated by that first commit, and gives no date of writing.

3. THE SOFTANZA FORM

Four steps, in the form the study of the record gives them: the AI agent proposes, the workbench rehearses, the court judges, and only a governed actor commits. I will show each in code, and then the cases where a step does not hold.

May this agent exist? Before an agent runs a single tick, the governance asks the question, and it asks it when the answer is free. An agent that declares no coverage, one sentence saying what it covers, is refused. An agent that declares no reversibility class is refused too: reversible, compensable or irreversible. Both refusals are in the engine's own sentences, and the door of the former host language repeats them. What does it say to an agent that declares no coverage, and to one that declares no reversibility class? Run it:

```
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
}
```

✓ ran: printed what this block promises

The first refusal ends on its law: nothing schedules what nobody has stated the reach of. The number it carries belongs to the library's own list of laws, which this book does not print. With both stated, the third call answers `1`.

Reversibility is the sixth contract of the governance, beside risk, authority, commitment, decommission and lineage. It is orthogonal to risk: an act of small risk that cannot be undone deserves more ceremony than a large one that can.

The harder an act is to undo, the more trusted the code that does it must be. Which postures may perform which classes of act? Here is the whole table, printed by a loop of the book's own:

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)
```

✓ ran: printed what this block promises

Read down the rows: trusted code may do all three classes of act, external code the first two, sandboxed code only the reversible. The last line asks for the refusal and prints its sentence.

The agent runs, believes it acted, and the disk does not move. An AI agent is a file and a few functions. Its function writes a price list. It is given a workbench, and the function reaches the bench as an ambient object during one cycle, with no change in its signature. The skill verifies its own work and the fact is learned. After one cycle, does the real file exist? Ask the disk, then the bench:

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")
```

✓ ran: printed what this block promises

The file lives in the twin. The agent believes it acted, and the disk did not move!

Two lines decide who may act. `posture: trusted` is in the file, because a `ring:` clause must carry a posture. `StzAllowAgentFunction` is a line of the platform's code, and not of the agent's file, and it decides which functions a file may call.

The only export is a plan, and it narrates itself. The tick exports toward reality a plan and nothing else. The plan says what it would do before anything is attempted. A language model set as its executor is refused before any attempt: `actor 'eager-llm' cannot commit -- it lacks the 'effectful' capability (required by operation 1)`. A named actor under a scope that does not cover the path commits nothing: the disk still did not move. The same actor under a scope that covers the path commits once, and the audit trail reads `[ [ 1, "committed", "write_file", "shop-committer" ] ]`. Who may commit, and when does the disk finally move? In this block, reality changes here and only here:

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)
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

I replaced the absolute path in the narration so the output is stable. A plan embeds the path of the file it would write.

The court at load. Is the table above enforced when the file is loaded, before the first tick? Take the same file, make the agent irreversible, and make its code sandboxed:

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)
```

✓ ran: printed what this block promises

The court answers with the table's own sentence: `a 'sandboxed' posture does not cover 'irreversible' work`. The same rule, at load: the file is refused before its first tick!

A proposal gone wrong. A tidy-up that deletes every file of a folder, driven here by hand through the twin as an agent would drive it, proposes thirty deletions. The plan lists thirty operations and flags thirty as risks. A language model that asks to commit is refused. And when it calls `Execute()` anyway, how many files are left? Count them:

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
```

✓ ran: printed what this block promises

this block touches files, the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

Thirty! The AI model asked, was refused, called `Execute()` anyway, and nothing on the disk changed.

The site shows this with a larger course, in its education demo of 30 September 2026, which I did not run for this book, and prints a count of six hundred and ten. That figure is the site's, quoted with its source and date. I print thirty, because thirty is what this block runs.

A person can argue with a plan. The plan is a medium. A reviewer rejects one step, with a reason. What do the other twenty-nine steps do, and what is left on the disk? The block prints the counts, not the name of the file:

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)
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

A named actor under a scope commits the other twenty-nine (29); one is skipped (1); and one file is still on disk (1), the file of the step the reviewer protected.

One rule, two layers, the same words. The refusal that chapter 17 showed at the graph, and that the file showed above, is also enforced in the engine's own tick loop, at registration. It is enforced in the sentence of the graph rule, no-llm-effectful. An AI model with the capability to produce effects is refused. One without it registers like any other slot, a slot being the place in the loop that one agent takes:

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()
```

✓ ran: printed what this block promises

The engine refused the summarizer, in the words of the graph rule! The arguments after the first, which is the agent itself, are, in the order the guard gives them: the kind (1 is an AI model), the reversibility class, the trigger and the capability. What remains is the priority, the interval in milliseconds and the coverage sentence. They are the engine's low-level names, and I show them so you can see the refusal come from the engine, in the same words.

An agent of the estate's own roster, rehearsed. Until 22 August 2026, an agent of the estate's own roster, the one that rolls the ledger, moved a real journal file directly, on a tick. It had no plan, no scope and no committing actor.

A ledger is a record of what happened, kept in the order it happened. This agent rolls the estate's own ledger. Chapter 21 has the security ledger, which the court writes and never the thing it judged, a hash chain in which an edited line is named. Chapter 23 has the constellation's ledger of crossings, with their timings. A register is something else: a form a reply can take, or Appendix A.

The doctrine had been written before it was true, and the record caught it the same day. Since then the function that writes for that agent has a rehearsing path. Inside the tick of an agent that holds a workbench it writes into the twin. Outside one, in the migration state the live estate still runs, it is still the direct write. The declaration below is my own, written for this book; it is not the live roster file, which I did not run. Given a workbench, the agent relocates the file in the twin. Two skills fire and verify; the real journal file is where it was; no archive exists; the tick has exported a plan. And only the committing actor, under a scope, moves the file:

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)
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The count of seven operations is what this run measures. The library's own guard only promises at least five.

The blocks have shown what the world does; section 4 says what I refused to build instead.

4. THE DECISION, AND WHAT I REFUSED

I refused safe agents. The doctrine's reason is that the field's safety tries to make a fundamentally unpredictable actor trustworthy. I hold this reason as a view, not a result. The doctrine's image is that asking whether the AI agent might escape is like asking whether a chess engine might move a piece on a physical board. It says the agent's tool surface contains no operation that touches the disk, a database or the network. That is the doctrine's claim, and section 6 gives its limits.

I refused a permission check as the boundary. It is not a check that the agent might route around. The powers are separated: creativity without authority, authority without creativity.

I refused a second workbench for machines. The sandbox is the same workbench that the human programmer already uses. The framework was conceived for human programmers, and everything designed to liberate one is exactly what is needed to contain an artificial one.

I refused to leave the gate optional. The ruling of 22 August 2026 says the registration gate is doctrine, and that opting in is a migration state, not the design. The record is honest about the finding behind the ruling: where the doctrine of agents that cannot hurt you bound that day was nowhere. The safe world existed, and the agentic folder did not mention any of it. The ruling closed it the same day.

I refused a stub. Asking for a reserved AI agent raises. The architecture says that this is the law working, and not a defect.

I refused to drop quietly. In the engine, a send to a full inbox is refused with a code and counted. That is tested in the engine's Zig code, in a test I did not run, and no block here and no narrated guard shows it. When the run queue is full, further visits are dropped and counted. The loop also states what it cannot see: the inside of a callback, the clock, and whether a coverage statement is true.

I refused to let a machine be the last resort. The interface constitution of the estate has a right, the sixth of six: the Right to a Human. Wherever a machine serves, a person is reachable, clearly, and without penalty. It is enforced by judgment, a person's, and not by a guard that runs; the library does not mention it. The constitution holds it as the human end of the loop: it is rule 102 of the constitution (Appendix C).

Section 5 names the guards that prove what the blocks showed.

5. THE COURT

`safeworld_narrated` has seven scenes: reversibility as the sixth contract; the registration gate in the engine's sentences; the table of posture against reversibility; the file's court at load. Three more follow: what the file declares landing in the agent's governance; the workbench, from the disk that did not move to the audit trail; and the roster's agent rehearsed and then committed. `agentfile_narrated` proves that an agent is a file judged at load, with one refusal per gate, and that a folder is a deployment. `agentloop_engine_narrated` proves that the engine loop states its coverage at run time, that registration is a gate, that one rule gives the same words in two layers, and that an over-full queue drops visits and counts them. `containment_drill_narrated` proves a drill against a real target on loopback: a model investigator proposes the containment and commits nothing; a human commits, and containment holds when verified from outside. In my run of 8 October 2026, the drill read 443 ms to detect and 61 ms to contain, on this machine. The book's data does not hold that run. The site prints other numbers from its own run, and the threat model others from its own. The three differ because the drill measures a clock. The limit of the drill is one attack shape, on loopback, on one machine, run on demand. `virtual_system_twin_narrated` proves the twin: a rehearsal touches nothing real; the plan narrates and flags its risks; a scope refuses; a reviewer rejects a step; snapshots roll back.

The guard `roster_narrated` is not in the front matter, because it cannot run from a snapshot of the library: it loads the estate's roster files from a sibling repository. The rehearsal of the roster's agent is proved by `safeworld_narrated`, whose seventh scene is the last block above.

Section 6 turns to the field, and to where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The record's own table is honest, and I carry its rows in prose.

The field leads in connectors: chat, drives and trackers, hosted vector stores, streaming interfaces. They are reachable from Softanza and not paved.

The field leads in retrieval over open corpora. Softanza wins on a curated domain and loses on the open web.

The field leads in frontier reasoning. That stack orchestrates the largest models on every call.

The field leads in ecosystem and hiring. There is a labour market for that stack, and none for this one.

When to reach for which, after the narration: the mainstream stack, when the task is open-ended language work over material you cannot curate. Its examples are summarising arbitrary documents, conversational products, research over the open web. The Softanza layer, when actions have consequences and someone will eventually ask why it did that. Among those cases, in the narration's words: "regulated or audited work, anything that runs unattended, and anything that must keep working offline at zero marginal cost." The two are not exclusive, and an AI model can sit inside a Softanza system as a proposer.

The twin is a model of reality, not reality. The framework asks for fidelity and honesty, and the plan re-validates against reality before it is committed. After the real file is deleted behind the twin's back, the validation says the target is absent in reality. I saw it warn, and did not test whether it blocks the commit.

And the largest honest loss. The threat model puts *code already running in the process* out of scope: it can do anything the process can, wipe the ledger or read a secret. The workbench contains what is written through the bench. A function that does not ask for the bench is not confined. Reads are free: the workbench's own header says reads are sensing and stay free, and that only writes are confined to the twin. The architecture says that the escape is not a defect to be closed by removing it. It narrows it with a declared posture, and with code, not the file, deciding which functions an agent may call. I did not run a trusted function that bypasses the bench: I read this limit in the architecture, the threat model and the workbench's header.

7. THE LIMIT

BUILT	reversibility as a declared contract and a registration gate; posture set against reversibility, at the door and at load; the workbench, the plan, the reviewer's rejection, the committing actor under a scope; the containment drill; the engine's bounded slots and queue (the bounded inbox is tested in Zig only, not run for the book)
DESIGNED	the agent's side of the negotiation over a plan; the examination hall; branches as parallel minds; twins for databases, interfaces and networks
ABSENT	a plan with no executor that is refused; a ranking of a plan's risks; code already running in the process, which is out of scope; the Right to a Human as a machine check

What is built: reversibility as a declared contract, and the registration gate, in the door of the former host language and in the engine; posture set against reversibility at the door and at load. The workbench is built, with a plan that narrates, flags, validates, takes a rejection and keeps an audit trail; so is the roster's agent rehearsed and then committed, and so is the drill. The engine's bounded structures are built: 64 agent slots, an inbox of 8 messages (tested in Zig only, not run for this book) and a run queue of 256.

What is designed: the agent's side of the negotiation, in which the plan reaches the agent but the conversation about the plan does not yet. The architecture says so, and the threat model repeats it. I searched the library's code and found no revising of a plan. The examination hall, which would certify an agent for a commit scope the way you certify a driver for a licence class: it occurs in documents and in no code. Branches as parallel minds: snapshots and rollback are built, and I found no branch call. Twins for databases, interfaces and networks: the framework names them. The twin of files is built, and so is the twin of the process environment, with its variables, its working directory and its queued spawn (chapter 24). The other three are not.

Each item below is a gap, found while running and reading.

- **An unset executor and an unset scope are unguarded.** A plan with no executor answers that no executor is set, and that a human at the keyboard is unguarded, and `Execute()` is allowed. The refusal of the block above applies to an AI model set as the executor. Likewise the scope check runs only when a scope is set: a plan with an executor and no scope is committed with no check on the paths it touches. So "only a governed actor commits" is exact when an executor and a scope are set, and a person at a keyboard may commit a plan that names none.
- **A plan flags; it does not rank.** The narration says that a plan ranks its own risks. The code returns one flag per destructive step, in the order of the operations, with a label such as DESTRUCTIVE. There is no score and no order of severity. The chapter says "flags".
- **The gate is opt-in at the host.** A program that hosts the library must declare that it uses the engine's loop; until the last such program in the library's own tree has declared, the default stays off, and a fresh program runs the older tick loop of the former host language.
- **The loop checks that a coverage statement exists, not that it is true.**
- **The drill's limits:** one shape of attack, on loopback, on one machine.

Six doors follow, the same chapter seen from six places.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your agent is a file and a few functions. It runs and believes it acted, and the disk does not move. You read the plan, choose the committing actor and the scope, and the court tells you in one sentence which rule refused you.
The analyst	Every proposal is a narrated plan of operations with the destructive steps flagged, and an audit line for each commit. You can read what the agent meant to do, step by step, without running it. The flags are by kind, not scored.
The designer	The plan is the one artefact that crosses from the agent to the person, and a person can reject one step. The Right to a Human says a person is reachable wherever a machine serves, and the right is held by judgment.
The decision-maker	You declare what each agent covers and how reversible it is before it may exist, and you decide the scope and the actor that commits. An AI model is structurally unable to commit, even when fooled; the threat model assumes it will be. The hardest part is the one the design leaves to you: which functions code allows an agent to call.
The educator	A learner's agent that tidies a course folder by deleting everything is a lesson in what a plan is. She sees what it would have done, and the files are still there.
The agent	You may rehearse without limit, inside the twin. Your only export is a plan, and you can be argued with over it. You may not commit. Today the argument is one-way: you do not yet receive the reviewer's objection and revise.

9. EXERCISE

Choose a folder of ten files you could lose. Write an agent file for a tidy-up job on it: what it covers, how reversible it is, one skill. Before you run anything, write under each line the promise you expect: how many files exist on disk after one cycle, how many operations the plan holds, and how many of them the plan flags as risks. Then write what happens when you set a language model as the executor.

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)
```

✎ not run: an exercise, written for you to run

Run it. Then give the plan to a committing actor under a scope that does not cover the folder, and write the promise again. If a promise breaks, do not change the promise: explain the output first. A good learner is surprised once. The plan flags its destructive steps and does not rank them, and a plan with no executor at all is allowed.

I left this one in your hands, on purpose. Take the block that committed the price list. The six lines that built `oScope3` and committed under it become these three, and the lines after them stay:

HARO

```
oPlan3 = oAg.GenerateUpdatePlan()
oPlan3.SetExecutor(PIActor("shop-committer"))
oPlan3.Execute()
```

✎ not run: an exercise, written for you to run

Run it, and ask the disk what the plan did, then ask the audit trail. I do not tell you what you will find. The page says why in section 7, in the first item of its list of gaps; read it after you have looked, not before. When you can say why "only a governed actor commits" holds for this plan or does not, you will know what that sentence depends on.

Chapter 21 is about the court itself, in four of its forms: promises, perception, security and performance. Its principle is that a claim is only as good as the court that can refuse it, and that where no court can see, the record says so.

WHERE THE EVIDENCE LIVES

NARRATION	stz-agents-that-cannot-hurt-you-narration.md
NARRATION	stz-service-virtualization-code-first-subscribe-later.md
ON THE SITE	agentic
ON THE SITE	security
RULINGS OF THE STUDY	12.18, 12.19, 12.22, 12.23, 12.29

PART IV · CHAPTER 21

The court: promises, perception, security and performance

DRAFT

Run on 2026-10-15 12:59 UTC+01:00 inside the library at commit 93d7a39bd: 7 of 7 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the court of chapter 5, which judges whoever wrote the code. I show four courts, each by something it refuses: promises, perception, security and performance. I walk the traveller over a file of three promises, and ask the record how many of eight musical traditions a person has heard. I edit a sealed ledger and read which entry the chain names, and I let an AI model propose a performance plan it may not commit. I say what I refused, seven things in all, among them the word "narration" for what cannot lie, a second assertion library, a sixth state for a stop, a guarantee without a guard. The principle holds the four courts together: a claim is only as good as the court that can refuse it, and where no court can see, the record says so.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Four practices of the field are four courts, and they are rarely seen as one.

The first is the example that runs. Python's doctest runs the interactive session found in a docstring and compares the printed text. Rust runs the examples in its documentation, and Go has testable examples with an expected output in a comment. The example is the specification. Its weakness is that the comparison is text against text, and the harness decides what a failure is.

The second is the snapshot test. A screenshot or a rendered tree is stored and compared on the next run. It proves that today's picture equals yesterday's.

The third is the audit against a list. The field's lists are OWASP's, NIST's and the compliance frameworks'. For detection and evidence, the toolchain is the log aggregator, the rule format, the orchestration of responses.

The fourth is the instrument. Micrometer, OpenTelemetry, Prometheus and the profilers of every runtime measure a program from the outside. All four are general knowledge, not a study of the record, except where I name the file.

2. THE FIRST PRINCIPLE

A claim is only as good as the court that can refuse it. And where no court can see, the record says so.

The record has it in two sentences that meet. The threat model: every guarantee names the guard that proves it, and a guarantee without a guard is listed as an open risk. And the perception law: a comparison proves sameness and never rightness, and if a person is the last consumer, a person is the last gate. Under both lies the doctrine that the library repeats in every plane: expression is free, and admission is governed.

3. THE SOFTANZA FORM

Chapter 5 told the court that judges whoever wrote the code. This chapter has four courts, one per kind of claim. I show each by something it refuses.

THE COURT OF PROMISES

A promise in this book is the line `#-->` under an expression, and it is a claim about a stop of the program. The tool that reads it, called the traveller (it belongs to the plane named Testoor), says a promise is kept, diverged, unreached or unjudged. A promise that argues with itself, prose after a value, is never accused. What does the traveller say of a promise that is true, one that is false and one that argues with itself? Run it:

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")
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

It says `kept`, `diverged` and `unjudged`, and the exit code is `1`. That code is the runner's, and not the language's.

A promise can be that the line raises. That is read as a raise that was promised. When it happens, how does the tour end? Run it:

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")
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The promise is `kept`, the end is `finished` and the exit code is `0`! The language process ended with an error, and the tour has finished as it said it would.

Every rule of this judge was learned by watching the harness accuse the library falsely. The guard's header says it. A short promise `6` once hid inside a later `[ 6, 28 ]`; that is why there is a rule that a short lie may not hide in a later list, beside the rule on the spellings a true value may print as. This book's own judge is a copy of the site's judge and not the traveller. It has fewer rules than the traveller, and the one it keeps is the one this book stands on: the build fails when a block does not keep its promise.

THE COURT OF PERCEPTION

The record's sentence is that a comparison proves sameness, and that it never proves rightness. The library holds the case. A bar plot drew vertical bars for horizontal data while all of its seventy-three examples went on passing. This is why the plots now compare the picture exactly. That makes sameness strict; it does not make it rightness. Rightness is a person.

So a declared world carries its perception state. Eight musical traditions are declared as data in the library. How many of them has a person heard? Ask the record:

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
```

✓ ran: printed what this block promises

It answers 8, 7 and HEARD: eight declared, and seven of them unperceived! Only one is heard, and that one's record carries a name, a date and a verdict. The counter in this block only counts the unheard; no counter decides the verdict. The one tradition that was heard was heard by me on 26 September 2026, through five original examples, and my verdict was "far from being qualified". Which part fails, whether the tuning, the phrasing, the rhythm or the timbre, I have not yet said. Those are my words, and they are the one recorded perception that is a refusal.

The law is the estate's, meaning my whole set of projects, adopted on 25 September 2026 on my instruction. It requires that the record names the perceiver and quotes the verdict. Its third requirement is the one I take with me: unperceived is a state, not a failure. It refuses two things as well. It does not say that a human must gate every commit, because it binds at the granularity of a phase. And it does not say that it generalises from one plane without further evidence, and it was held for a month on that account.

The gallery of pictures is where it is unmet. The library holds ninety-five judgements of pictures and text sections, in three index files, committed on 5 October 2026. Every one is by an AI model, reading the picture or reading the text the program printed. None has a human name. The first index says it in a sentence: no human has looked at these pictures. The register of findings ends with a line I repeat: a human verdict on a sample is still owed. The record holds no human verdict on any gallery picture. It stays unperceived.

THE COURT OF SECURITY

The security plane starts from a claim. The threat model assumes that a hijacked language model will get through, and makes it harmless. The AI model never holds the capability, and so an AI model that is fooled still cannot act. The document does not try to detect every malicious prompt.

The evidence of what happened is a hash chain. Edit any line after the first, and the chain says which entry broke. If I edit one, which entry does it name, and does a wrong key draw the same accusation? Run it:

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")
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The verifier answers `chain intact over 2 entries`, then `the SEAL does not match this key`, then `the chain breaks at entry 2`. The chain named the break! A wrong key and an edited line are two different accusations. Notice which entry I chose to edit: the second. An edit to the first entry is not caught. The verifier checks each entry's text against the digest of the entry before it, and the first has none before it. That hid my first attempt. The narration states the limit in its own words: this is not protection against an attacker already running code in the process.

A secret is the same law applied to a key. Its only public face is its descriptor. A hand that may commit can read it. What does the court answer when a language model asks? Run it:

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()
```

✓ ran: printed what this block promises

It answers `Refused: actor 'planner' (posture sandboxed) may not reveal secret 'deploy-key'`. The court refused the AI model, and the ledger counts 1 refusal, though no line of the block records anything to it! This is chapter 17's law, that a model does not hold the capability to produce effects, applied to a key, with the record added.

What does the table of guarantees tie to a guard? The meta-guard of the threat model checks that the document exists with its sections, and that every guarantee row contains a proof. A proof is a guard's path, a fuzz run, a bill-of-materials check or a runbook. It checks that every guard path in the rows exists on disk, and that a planted fake row with an unproven guarantee is caught. It does not run the guards, and it does not check that a guard asserts what its row says. That is the security gate's job. So "each tied to a guard" is, exactly, "each names at least one proof, and every guard file it names exists".

At this commit the table holds 43 guarantees. The site says 38, on three pages. Both are true of their dates: the document had 38 rows on 30 September 2026, and the rows from 39 to 43 were added on 5 October. The site's number is the site's to update.

THE COURT OF PERFORMANCE

An instrument owes a number. The honest stopwatch gives a number from a clock that only moves forward. It excludes a pause, and it ships two spans of one trace in one batch for an observability collector. If I pause it for 200 milliseconds, does the pause count in the elapsed time? Run it:

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
```

✓ ran: printed what this block promises

The pause did not count: in the stopwatch's scene, `ElapsedMs() < 200` printed `1`, and `Record()[:running]` printed `0`, because the stopwatch had stopped. The second stopwatch joined the first one's trace: `w2.TraceId() = w.TraceId()` printed `1`, and the batch's JSON holds the trace identifier `2` times, for two spans.

Careful with what this does not claim. The narration says the joined stopwatch gets a new span. I checked: it gets the same trace, as the `1` above shows, and the same span identifier, which this block does not print. The sentence holds for the

server's echo, which I did not run, and not for the stopwatch. It is a request to the library.

The instrument is not the end of the court. Judgment and action live inside the program. A performance plan is proposed by a machine, and only an effectful actor commits it. What happens when an AI model asks to commit it, and then a person does? Run it:

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)
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The fleet here is a rehearsal double, any object that answers the same effectful verbs. The loop is proved against the double and not against a real fleet. A model asking to commit is refused before any attempt: `ExecuteOn` printed `0` and `CallCount` printed `0`, so the double counted zero calls! For the person, `ExecuteOn` printed `1` and `CallCount` followed with `1`. The same action is judged twice, and the audit keeps both judgments: `REFUSED scaleup 'nlp' by advisor -- actor is not effectful -- expression is free; admission is governed`, and `COMMITTED scaleup 'nlp' by mansour`.

The four courts have spoken, and now I say what I refused.

4. THE DECISION, AND WHAT I REFUSED

I refused the word for the thing it was not. The structure of this book and the closing assessment of the study call the articles "narrations that cannot lie". My own ruling of 7 October 2026 says that "narration" names the articles under the library's documentation folder, and not the executable suites. The site's runner annotates an article and no longer decides whether it is shown. The site's own record of 8 October 2026, run on another commit than this book's, says that of 135 articles 55 ran. Of the 423 blocks in those, 123 kept their promise. That count is the site's runner's, of articles on the site's commit, and not a count of the library's guards. So the articles can drift. What cannot lie are the executable guards, and this book's own build, which fails when a block does not keep its promise.

I refused a second assertion library. The promise runner's rulings refuse a second library, a rewrite of the test files, a coverage percentage before there are routes, and a mock written for a platform's production. They refuse the word "verified" on a perceptual claim without a name and a sentence.

I refused to fold unjudged or unperceived into pass or fail. A stop has five states and no sixth: kept, diverged, unreached, unjudged, unperceived. A runner that folds unjudged or unperceived into pass or fail is wrong. I hold this as a view, not a result.

I refused to analyse tampered evidence. A tool that verifies, warns, and then analyses anyway teaches its users to ignore the warning.

I refused to detect everything. Detections are declared. The sentinel speaks on transitions only: quiet, one alarm after five wrong passwords, and nothing new after a sixth, since the story did not restart. It does not write its own firings into the ledger, because detections over detection events invite feedback loops.

I refused to compete with the field's ecosystem. The performance study of 29 July 2026 says the plane does not compete with Grafana, Datadog or the ecosystem of OpenTelemetry, and should not pretend to. It exports.

I refused a guarantee without a guard. The threat model's last rule is that a guarantee is added only together with its guard.

Now the guards that prove what ran.

5. THE COURT

`testoor_promises_narrated` proves each promise rule planted both ways. A lie is diverged; two lies are both caught; a short lie does not hide in a later list; the spellings of a true value are all kept. An expected raise is honoured; prose is unjudged; a program that does not compile is not diverged; a raise part-way leaves the later promises unreached. `threat_model_narrated` proves the meta-guard described above, including its negative sibling. `security_ledger_narrated`, `secretstore_narrated` and `secret_narrated` prove the chain and the seal, the store and the secret. `security_sentinel_narrated` proves the edge-triggered sentinel. `stopwatch_narrated` and `tracing_tail_narrated` prove the instrument and the batch. `governed_loop_narrated` proves the plan, the preflight, the commit and the audit.

The build counts 11 assertions in `threat_model_narrated`, a static count of its assertion calls. The others are counted by the build too, and not run here. The performance plane has twelve guards. I do not print their sum, because the build counts them.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The record's own honest sections, each loss beside its win.

In performance. A time-series database, a query language and dashboards are the field's today, and absent here. By design, the plane exports. Its sampling answer is bounded rings that drop the oldest; head and tail sampling are what the field does better today. Distributed context reaches HTTP today; going past it is the next step. The ecosystem of integrations and auto-instrumentation for every framework is the field's. The profiler is cooperative: it sees what you bracket, where a native profiler sees everything. The win it claims is that judgment and action live inside the program: budgets that fail the build, tuning that passes governance, monitoring that prices itself.

In security. The ledger is a bounded ring, and bounded means forgetting. Telemetry from the host, the kernel and the network is absent; the plane reads what the program's own seams emit. Detections are declared and not learned; learning them is the next step, and one detection of rates on 30 September 2026 already covers part of it. The record's own comparison is that a rule corpus of the field's format is larger today than the eight detections shipped, and the field leads here. The workflow of a security operations centre is the field's today. The largest gap, in the record's words, is that a detection can only see what a seam emits. The win is that the event is typed at birth, where the door refuses, so no parser stands between what the program knew and what the detector judges.

In promises. The field's doctests sit in the host's toolchain and answer pass or fail; that is general knowledge. The house's judge has five states. Where the field leads is instruments: the launch memo of Testoor, the plane that holds the traveller, counts coverage instruments of any kind at zero today, and route coverage is the next step.

In perception. The field's visual snapshot services are mature and automatic; that is general knowledge. The house's law says a perception does not replace a number, and no number replaces it.

The study of the record puts the tone I try to echo. A grid of two hundred and eighty-six ticks against two hundred and sixty-nine crosses is, in its words, the shape of a brochure.

Now to where the chapter stops.

7. THE LIMIT

BUILT	the promise judged as a claim about a stop; the security plane with its ledger, sentinel, drill, response and secrets, and its table of guarantees tied to guards; the performance plane with its instrument and its governed loop; the perception law, adopted
DESIGNED	routes, the clock and weather, and perception for the promise runner; the plugin plane; the negotiation over a rejected plan
ABSENT	a human verdict on any picture of the gallery; seven of the eight musical traditions heard by a person from them; a real vault and a real sandbox as proofs; a time-series store

What is built: the promises as stops; the security plane, with its ledger, sentinel, drill, response, attestation and secrets; the performance plane from its first phase to its last; the perception law, adopted. And the declared verdict tables for the singing voice and the eight universes. The table of guarantees holds 43 rows that name 57 distinct guard files, none missing.

What is designed:

- The promise runner's routes and map, its clock and weather, its excursions, perception and mutation, and its inherited tours. All are planned. The state *unperceived* is named in the runner's header, and the reader assigns none of them.
- The negotiation over a rejected plan, which the threat model calls design only.
- The plugin plane: plugins are not shipped, and foreign code is not yet governed.

What is absent:

- **A human verdict on any gallery picture.** Zero of ninety-five. The law's second requirement, a name, a date and what they said, is unmet there. The chapter says unperceived, and does not say verified.
- Of the eight musical traditions, seven unheard.
- A real vault and a real sandbox as proofs. The vault resolver is tested against a stand-in, over real HTTP. The cloud key-management adapter is absent.
- Paths on POSIX systems, unverified, until a Linux machine is available; scheduled fuzzing; continuous integration; signed commits, which I deferred for the development phase.
- A time-series store. A coverage percentage, which is refused until there are routes.

Three more things belong here, found while running the court.

The drill's numbers differ by source. The threat model's measure of responsiveness comes from three runs. It is a detection in a little over three hundred and fifty milliseconds, and a containment in a little over fifty. The site prints 395 and 50, from a replay of 1 October. They differ because the drill measures a clock. One attack shape, on loopback, on one machine, run on demand. Whichever number you print, you name its source.

A possible defect in the promise judge, which I report and have built nothing on. A file whose first promise is a lie and whose last is a promised raise reads the lie as unreached and the raise as kept. The tour is broken, with zero diverged, and the exit code is 1. I reproduced it twice. Each rule alone behaves as the guard plants it. Whether a divergence above a promised raise should read *diverged* is a question for the Testoor plane, which holds the traveller.

Counts that disagree between sources. The study's closing says 387 narrated guards. My `find` on the pinned snapshot says 529 files named so. The build's own count of guards is the one to use, and neither of the two.

Six doors follow, one for each reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your examples already carry their expected output. The traveller reads them as promises and says kept, diverged or unjudged, and prose that argues with itself is never accused.
The analyst	The threat model is a table you can read: each row with the guard that proves it and its honest limit. The performance judgment is declared before the code runs.
The designer	A picture or a sound is a claim only a person can close. The gallery says an AI model looked and no human has. Your verdict is a named line in the record.
The decision-maker	The question who decided, and what is written down, has a mechanical answer in both planes. The same action is refused for the AI model and committed for the human, both in one audit, and the evidence is sealed so that an edit after the first entry is named.
The educator	The tutor's rule meets the court. Predict the verdict of a promise before the traveller walks it.
The agent	You may propose a containment plan or a tuning plan and you may commit neither. Your refusal is itself a recorded event. Do not tell a person that a picture is right. Say "unperceived".

9. EXERCISE

Predict, in writing, before you run anything. The file below holds three promises. Which state will each one have?

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")
```

✎ not run: an exercise, written for you to run

Run it. The states are kept, diverged and unjudged. Then add a fourth line that raises, promise it with `#--> ERROR:`, and predict first how the tour will end. Do not change your prediction to match what you see. Explain the output first.

I left this one in your hands. Here is the raise block of section 3 with one change: its first promise is now a lie.

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")
```

✎ not run: an exercise, written for you to run

Run it. Read what the traveller says of the lie, what it says of the raise, how the tour ends and what the exit code is, then compare with what each of the two promises gave alone in section 3. Section 7 says why I cannot tell you: it is a possible defect in the promise judge, which I reported and built nothing on, and I left it unfixed on purpose. When you can say what the court answers here, you will know the question that section 7 hands to the Testoor plane.

Chapter 22 is *Compared, honestly*. Its principle is this: a comparison is worth nothing until the other side's version and the date are on the page, the other side has been run, and the loss is printed beside the win.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	security
ON THE SITE	offering
RULINGS OF THE STUDY	12.13, 12.16, 12.19, 12.20, 12.22, 12.26, 12.29

PART IV · CHAPTER 22

Compared, honestly

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 5 of 5 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I compare Softanza with the field, and I say how a comparison must be made before it may be printed. I show five pairs, the same task run on both sides, with the other side's version and the date on the page and a line where the other side leads. I say what I refused, the grid alone first, and which guards prove the blocks, since the comparison court is proposed and not built. I print the list of what the field has and Softanza lacks, beside the wins. The principle is this. A comparison is worth nothing until the other side's version and the date are on the page, the other side has been run, and the loss is printed beside the win.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Comparison, in the field, is a genre of marketing with a few honest exceptions. The record names the genre: a "Compare" page for each rival, and a comparison block on each feature page. Its commonest form is the grid, with rows of features, columns of tools, ticks and crosses.

The honest exceptions are practices and not products. Benchmarks published with the method, the machine and the build mode. And sections that say what the other tool has that we lack. That is general knowledge, not a study of the record, and I name no tool.

2. THE FIRST PRINCIPLE

A comparison is worth nothing until the other side's version and the date are on the page, the other side has been run, and the loss is printed beside the win.

It is built from three sentences of the record. My study of the record, of 7 October 2026: every claim about another tool is cited, dated and run, and every loss is published with the wins. The estate's study of a browser runtime, run on 8 August 2026: losses are published with the wins. And the site's page of coverage: a platform that shows its gaps is believed on its strengths. My request, as the study reports it, is a comparative tone that is "a motor force, to learn from the best and beat them when possible". The study adds: with "when possible" measured.

3. THE SOFTANZA FORM

Chapter 3 said "I refused the grid alone". This chapter is the method that replaces it, with its evidence. What did the record measure about the grid? The study scanned the library's text for tables whose header names Softanza and at least one other tool. It found sixty-four tables, six hundred and fifteen rows, in thirty-one documents. In them, the column of Softanza carries 286 ticks, 2 warnings and 2 crosses. The other columns carry 204 ticks, 207 warnings and 269 crosses. The study's judgement is that this is the shape of a brochure, and that the record itself does not trust it. I could not reproduce those figures: a narrower scan of my own, with a header rule I wrote, found thirty-seven tables in twenty-nine documents. The study does not print its header rule. I cite its numbers as its own, and dated.

The form that replaces the grid is a pair: the same task, both sides run, the other side's version on the page, and a line where the other side leads. I show five. Four were run on both sides, and the fifth, block five, on the Softanza side only. The runs were made on 8 October 2026, between 22:24 and 22:34, on a Windows machine, and the book's build ran the Softanza blocks again on 9 October 2026 at 05:32. All kept. The other side is Python 3.13.15, and Node v22.20.0 with its V8 12.4 engine and ICU 77.1. R and Julia are not installed on this machine, and no pair uses them.

One: an integer is exact at any size. How many digits does the product of two numbers of twenty keep? Run it:

HARO

```
oN = new stzNumber("12345678901234567890")
? oN.MultiplyByQ("98765432109876543210").Content()
#--> 1219326311370217952237463801111263526900
```

✓ ran: printed what this block promises

Forty digits, exact! The other side, Python 3.13.15, prints the same digits, `1219326311370217952237463801111263526900`, for `12345678901234567890 * 98765432109876543210`. Python's integer is exact by default, and Softanza is level with it. The Softanza line is the longer one: a literal this long cannot be a native number, so it is given as a string. The other side, Node v22.20.0, prints `1.219326311370218e+39`, because its default number is a 64-bit float. It is exact only when you ask for it with the suffix `n`, a second kind of number. So where is the win? Over JavaScript's default, and not over Python.

Two: exact decimals. Is `0.1` plus `0.2` equal to `0.3` here, and does a price times a rate keep its cents? Run it:

HARO

```
? (new stzNumber("0.1")).AddQ("0.2").Content()
#--> 0.3
? (new stzNumber("19.99")).MultiplyByQ("0.15").Content()
#--> 2.9985
```

✓ ran: printed what this block promises

It answers `0.3` and `2.9985`: the cents are kept! The other side, Python with floats, prints `0.30000000000000004` and `2.9984999999999995`. Node prints the same two, because JavaScript has no decimal type. Python's standard `decimal` module, imported and fed strings, prints `0.3` and `2.9985`: level with Softanza, and with the same cost per line. So Softanza wins on the default, since a number made from a

string is exact decimal, and is level with the one library that does it deliberately. The guard's comment says why: the value is stored as a string, and the computation is done on scaled integers by the engine. Its limit is in the record's words, and one of them is stale. The rungs for rationals and for complex numbers are named in the plan and, in a comment of the guard of numbers that are not lost, "not built yet". The snapshot holds a narrated guard for each, `numeric_rationals_narrated` and `numeric_complex_narrated`, so that comment is out of date and I do not rely on it. I did not run those two guards.

Three: sorting accented words. Here the field leads. Softanza sorts by code point. Where does that put `étoile`? Run it:

HARO

```
? @@( Q([ "zèbre", "étoile", "eau", "zoo" ]).SortedInAscending() )
#--> [ "eau", "zoo", "zèbre", "étoile" ]
```

✓ ran: printed what this block promises

It answers `[ "eau", "zoo", "zèbre", "étoile" ]`: `étoile` last, after `zoo`! The field's locale-aware sort puts `étoile` between `eau` and `zèbre`. The other side, Python's `sorted`, prints the same list: level, and both wrong for French. With the standard `locale` module set to `fr_FR.UTF-8`, sorted with `locale.strxfrm`, Python prints `eau, étoile, zèbre, zoo`. That worked on the machine of the run. The locale must exist on the host, and the call is process-wide. Node, with `Intl.Collator('fr')`, prints the same French order, `eau, étoile, zèbre, zoo`, with a built-in library. So the field leads: JavaScript with one argument, Python with the host's locale. Softanza has no collation. The culture plan of 18 August 2026 says the sorting is the engine's ordinal sort. And the coverage page rates internationalisation partial, against deep for two other platforms, with a note: a deliberate bet against the big international library.

This block is a pinned gap. Its promise records the library's present answer, so the page fails the day the library gains collation. That is by design. The culture plan's own rule says that pinning a known gap is legitimate only when the assertion fails

the day the gap is fixed, and when the label says so.

Four: Turkish uppercase. The field leads. What does the method answer when it is given the Turkish locale? Run it:

HARO

```
? Q("istanbul").UppercasedInLocale("tr_TR")
#--> ISTANBUL
```

✓ ran: printed what this block promises

It answers `ISTANBUL`: the locale was accepted and ignored! Python prints `ISTANBUL`: level, since the standard library has no locale casing. Node, with `toLocaleUpperCase('tr')`, prints `İSTANBUL`. Turkish needs the dotted capital. This is the loss most worth printing, and it is not that the engine lacks Turkish. It is that the method takes a locale and, today, answers with plain uppercase without saying so; honouring the locale, or refusing, is the next step. The source comment says: plain uppercase when it has no rule. The culture plan measured the same on 18 August 2026 and called it the code equivalent of the fake progress bar. Its law is the reason for this chapter: no facades. An operation either honours the culture it was given, or refuses. I hold this as a view, not a result, and the culture plan's law says the same.

Five: an instruction in your language. A natural-language instruction is dictionary-driven and local: no network, no AI model, no key. What does it do with the sentence, and what did it understand? Run it:

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
```

✓ ran: printed what this block promises

It answers `[ 3, 1 ]`, and says what it understood: `create a list with [ 3, 1, 3 ] -> remove duplicates`. I did not run the other side. The field's equivalent of "take this sentence and do it" is an AI model call. What it needs is a model. A hosted one needs a network, an account, a key and a per-token cost; a local one needs a downloaded model and a GPU or a patient CPU. It needs a prompt, and a check that the answer is what was asked, and the answer is not guaranteed to be the same twice. The site's own pair of this kind writes the other side as a comment, which says that there is no counterpart without an external language model.

One more loss, which is not a block. A string of a family emoji, five code points, joined by joiners, shows where the public string does not yet expose what the engine knows. The counter class answers `1` for the grapheme count; `NumberOfChars` answers `5` and `NumberOfBytes` answers `18`; and `GraphemeCount()` on the string itself raises, because the method is not defined. The engine knows, and the string does not say. Python's `len` gives `5`, and Node's `.length` gives `8`.

That is the form, five pairs and one more loss; what I refused to reach it comes next.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused the grid alone. Two hundred and eighty-six ticks in Softanza's column against two hundred and sixty-nine crosses in the other columns is the shape of a brochure.

I refused to keep a headline that does not reproduce. The estate's memo of 13 September 2026 re-measured the runtime beneath the library against a fast runtime of the field on six matched kernels. Matched means identical algorithm and sizes on both sides, each side its fast idiom, cross-language checksums asserted before any number, minima over repeated runs. The field leads every row, by a factor between 1.04 and 1.88. The memo says the claim to retire is that the runtime beats it, in the present tense, on x64. The headline was never re-measured after the spike, as the memo calls the runtime's prototype, became a product. The memo asked me to correct the runtime's own notes. The headline was withdrawn in the estate memo of 13 September 2026, and I say no more.

I refused a retraction that deletes its evidence. The runtime's own notes of 29 August 2026 retracted a published table in the other direction, a loss on a phone. The run had executed on the phone's slow cores while the peer ran on fast ones. The notes keep the table, struck through, and say that a retraction that deletes its own evidence is not a retraction. So the record has withdrawn a win and withdrawn a loss. The rule is to reproduce before you print. Printing losses is not the rule. The two withdrawals the record holds, a win and a loss, are what stand behind it.

I refused a comparison that cannot be redone. The hand-off packet of the same day, 13 September 2026, says the spike's kernels died with the working tree. That is why an outside session had to reconstruct the measurement from prose.

I refused a facade. An operation either honours the culture it was given, or refuses; blocks three and four print where the library stands on that today.

I refused the large international library. The culture plan's table says it is killed, a decision made and standing in the first layer. The cost is blocks three and four. The coverage page prints it as a deliberate bet.

I refused to compare on the tool's strongest ground. The estate's study of a browser runtime has three honesty rules. Each language gets idiomatic code; losses are published with the wins; every engine is a release build. It says, of its own page, that without them it is marketing. Its scoreboard has nine scenarios, four won and five lost. It is not a regression gate: its numbers are for learning, and not for gating.

The court that would hold these refusals comes next, and it is proposed, not built.

5. THE COURT

The comparison court is proposed and not built. The study's plan names it: a comparison row with no source or no date does not publish. Nor does a code pair whose other side did not run, or ran in a debug build. I found no file in the library or the site that implements it. So what exists? Four guards, which prove the blocks and bear on the method.

`numeric_exactness_narrated` proves exact decimal arithmetic, integers past 2 to the power of 53, the forty-digit product, and exact division when it terminates. `numeric_no_loss_narrated` proves that a number is not destroyed on its way in, and that the representation ladder is observable. Its closing comment still names the rungs for rationals and complex numbers as not built, and is stale (see block two). `semantic_resolver_narrated` proves that paraphrases resolve to canonical operations through the local lexicon, and block five's sentence is one of its cases. It does not pass at this commit: I ran it on 8 October 2026 and it stops with an error raised from a string-splitting function, before it ends. So the page counts its assertions and does not claim its verdict. `engine_is_optimised_narrated` proves that the engine is compiled with optimisation, by a time gate: it is the library's own guard for the release-build rule.

No guard asserts blocks three and four as pinned gaps. No guard exists for a code pair whose other side did not run. Both are requests to the library.

The court is not built, but the record already keeps the list of what the field has, and it follows.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The record already keeps the list of what the field has and Softanza lacks. I print it, by file and date, beside the wins.

Performance, against the field, 29 July 2026. No time-series store, no query language, no dashboards: by design, the plane exports. No distributed context beyond HTTP. Sampling that is honest, where the field's is finer today. Decades of integrations, which the incumbents own. The study's own words are

that on breadth the incumbents win decisively, and that the plane does not compete with them and should not pretend to. Four losses are struck through with the phase that closed them.

Incident analysis, against the field, 1 August 2026. Scale, retention and search, where bounded means forgetting. No telemetry from the host, the kernel or the network. A small set of shipped detections against a community's corpus. The workflow of an operations centre. And the unwired seams, which the study calls the system's largest honest gap, listed and not hidden.

The plugin study, 16 August 2026. The field is ahead on permission granularity, on dependency resolution, on distribution and signing. The study's header says it is analysis and not rules.

The culture plan, 18 August 2026. Every row was verified in that session by running it. Casing takes a locale four levels deep and does not yet honour it. Collation, digit shaping, non-Gregorian calendars and time zones are absent. The plan's status then was not started.

The estate's browser-runtime study, 8 August 2026. Four of nine rows won, five to the peers, and the fused loop opcode of one peer stays a design generation ahead.

The runtime against a fast peer, 13 September 2026. Six kernels, checksums first, the peer ahead on every row.

The string benchmark of 7 July 2026, and the polyglot battle, in a narration dated 18 April 2025: the former host language came last of seven in two of three rounds. It came last in the total too. Both carry caveats the later record has put on them. I did not re-run the benchmark. The engine's default build was a debug build until 26 July 2026, and the library's own build file says every shared library it had ever loaded was compiled unoptimised. So the benchmark probably ran unoptimised. That is an inference, and I state it as one. The same comment says that on string, list and matrix work the change buys only about fifteen percent. So the build mode is a caveat on the numbers and not an excuse for them. I could not read the build mode of the engine this book runs on, and the clock of the former

host language resolves ten to fifteen milliseconds; a fair re-run needs both read. So the benchmark is cited here as a dated record, with that caveat, and not as a present measure.

The coverage matrix. Thirty domains in seven groups, against four other platforms, read on 17 September 2026. Softanza's tally is seven deep, fourteen solid, seven partial, two absent, present in twenty-eight of thirty. I recounted it from the rows and it matches. The two absent rows are symbolic mathematics ("no computer algebra") and package, build and deploy ("a library, not a registry"). Among the seven partial rows is internationalisation, "a deliberate bet against ICU", and the neural row is "inference only; no training foundry". The stored tallies of two of the other platforms do not match their own rows, by one in two places. I print Softanza's tally only. The matrix cites a compass file as the source of the ratings. I could not locate the file in the snapshot, in the live checkout or in the estate's repository.

The graph comparison, a narration dated 26 December 2025. An AI model wrote it, and says so in its byline. Its column for Softanza rates itself with a cross on scalability and on transactions and with a warning on performance, and its conclusion lists where traditional solutions win.

Where Softanza stands apart, by the record, without a number: the SLA; the governed act; the typed event; the matrix's one row of governance and safe execution. Two more: exact numbers by default, against two floats; an instruction that resolves locally. And a method that prints each of them beside a loss.

That is the list; now I say where this chapter stops.

7. THE LIMIT

BUILT

five comparisons, four run on both sides with the other side's version and the date, and the fifth run on the Softanza side only; the four code pairs of the site, re-run, three of them on both sides; three field studies that end on what the field has and the library lacks, and a culture plan that prints its losses

DESIGNED — a comparison register as data; a comparison court whose rule is that a row with no source or date does not publish; both benchmarks re-run on the present engine

ABSENT — locale collation; locale casing; a grapheme count on the public string; a time-zone database; the benchmark of 7 July 2026 re-run; the source of the coverage matrix, which I could not locate

What is built: four of the five pairs, run on both sides, and the fifth on the Softanza side. The four code pairs of the site are built too. I re-ran them on 8 October 2026, three of them on both sides (the fourth has a comment for its other side). The outputs matched. So are the library's field studies of performance, incident analysis and plugins, with their loss sections, and the culture plan, which prints its losses. So are the harness of the string benchmark and the guard that the engine is optimised.

What is designed or proposed: a comparison register as data; a comparison court; the re-run of both benchmarks. The study holds all three. On 8 October 2026 I searched for them in the library and in the site and found nothing. The site, read on 9 October 2026, prints five rules for comparing, the first being "Cite or do not publish". It also prints a list of the nine rows where Softanza is partial or absent, with the platforms that are deep there. It says, in a line marked designed, that a register of comparisons is not held as data yet. It does not name the court.

What is absent: a *compared* column on the Atlas, on the register of innovations and on the narrations. Absent from the site: the field studies, the sixty-four grids themselves (it now gives their number and no more), both benchmarks and the runtime table. Absent from the library: locale collation; locale casing; a grapheme count on the public string; a time-zone database. And absent still: the re-run of the benchmark of 7 July 2026, and the source of the coverage matrix.

And something that is unreproduced: the study's sixty-four tables, six hundred and fifteen rows and thirty-one documents, which my scan did not reproduce, and the symbol counts.

The method is stricter than the record it is applied to. Older comparisons in the library predate its rules and do not yet meet one or more of them. The string benchmark probably ran on a debug engine. The polyglot battle has no versions and no dates inside its text, and says its times are relative to a specific computer environment. The graph grid has no versions, dates or sources for its cells. The cognitive-load grid of another design article makes claims of sixty to seventy percent with no method printed beside them. That article then ends on an honest positioning that names four things the library does not compete on.

Six doors follow, one for each reader of this page.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Copy a pair. Run both sides in your own versions, keep the output as it came, and add the line where the other tool is ahead.
The analyst	The coverage matrix is a buying aid only because it shows two absent rows and seven partial ones in its own colours. Read those first.
The designer	The loss goes in the same block and in the same type as the win. The stage word is part of the interface.
The decision-maker	Ask any vendor, this one included, for the other side's version, the date, whether it was run, and the line they lose. The memo of 13 September 2026 is a model of a headline withdrawn.
The educator	A code pair is a lesson in what a line costs. The learner writes the other side's output before running it.
The agent	When you repeat a claim about another tool, carry its version and date, and say if you did not run it. The register that would make this checkable by an AI agent is proposed and not built.

9. EXERCISE

Pick one task where you believe your own tool is ahead, and one where you believe the other is. Make a comparison that obeys the three rules: the other side's version and the date, both sides run, the loss printed.

HARO

```
? YourSide(the_input)
#--> the line your tool prints
```

✎ not run: an exercise, written for you to run

Under it, write as text: the other tool's name and version, the date, the exact line it prints, and one line of the form "here the other leads because...". A reader checks the three text lines, since a runner cannot.

A variant that a machine can judge: take the three runs of block three's other side, Python's `sorted`, Python with the locale, and Node with the collator, and predict, before running, the order each one prints. I will not tell you which order is right.

I left this one in your hands. Here is block three with its promise taken off:

HARO

```
? @@( Q([ "zèbre", "étoile", "eau", "zoo" ]).SortedInAscending() )
```

✎ not run: an exercise, written for you to run

Under it, write the promise a French reader would expect, the order the field's locale-aware sort gives in section 3, as a `#-->` line, and run it. I do not print the verdict. Read it, then read sections 3 and 7 to learn why this block is a pinned gap and not a fault of the sort. When you have solved it, you will know why block three promises the library's present answer and not the field's.

Chapter 23 turns to the architect, and to a world, a constellation and a platform. There the solution is declared as worlds, bonds and an envelope before any of it is built, and every crossing between them is governed.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	compare
ON THE SITE	code
RULINGS OF THE STUDY	12.10, 12.22, 12.29

PART V

The platform, assembled

23 · A world, a constellation, a platform, and the architect
24 · Rehearse, then commit
25 · Your users keep their medium
26 · The environment
27 · The engine beneath: three layers, one engine, Haro

PART V · CHAPTER 23

A world, a constellation, a platform, and the architect

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 10 of 10 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I declare a solution at three levels, a world, a constellation and a platform, and I name the one person who declares all three, the architect. I take the field's three habits first: the super-app, the context map and the platform team. Then I run a world that explains itself, a constellation whose crossings are refused one gate at a time, and a profile that a platform refuses to build when it is not sound. I say where the code stops: a world's truths are narrated and not judged, the shells are tiny, and the architect's certification is a proposal. The principle is this: the solution is declared as worlds, bonds and an envelope before any of it is built, and every crossing between them is governed.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Three habits of the field have each a home, and I take them from general knowledge, not from a study of the record. The record holds no field study of super-apps, of platform engineering or of context maps. Its only named field is one line of a design document: a famous messaging super-app, and its kin.

The first is the super-app and the mini-program. One large application embeds a runtime and runs small applications inside it, sharing its services and its store. The host is privileged, the mini-app second-class, and the isolation a sandbox.

The second is the bounded context and the context map of domain-driven design, of 2003. A large system is cut into models, each with its words, and a map shows how they relate. It is the field's nearest idea to a world and a bond. It lives in diagrams and in review, and not in a running registry that refuses a call.

The third is the platform team and the deployment declared as code. A team builds the internal base that others deploy onto, and tools declare machines before they exist. The person who works across the floors is, in the field, a software architect, a solutions architect or a platform engineer. It is a title held by a person, whose artefacts are diagrams and manifests.

For the scopes, the field's home is the compiler: the target of a build, and the check that flags a call the minimum platform level lacks. Each decides which system the code is for. None lets the program name that system at the call site and refuse the act.

The shared shape: the field draws the picture, a host, a map, a diagram, and keeps the judge outside it. In all three the program knows nothing of its own composition. This is my reading of the three, not a survey of the field.

Against that shape I set one principle.

2. THE FIRST PRINCIPLE

The solution is declared as worlds, bonds and an envelope before any of it is built, and every crossing between them is governed.

The sentences of the record are three. An application is a living world of meaning. Worlds, not mini-apps: a governed graph and not a host container. A Softanza architect models the deployment architecture first, as a declarative artefact, before a line of system feature code. The sentence above is built from the three and from the book's method, which is to declare and then govern. The three are the record's, put in my words; the sentence above is the chapter's own principle, in the book's voice.

3. THE SOFTANZA FORM

Three levels, then. A world is one application. A constellation is a graph of worlds. A platform is the envelope around a solution and the ground a constellation stands on. And one person declares all three.

A WORLD

A world has a being, a becoming, and a way of being met. Chapter 4 showed its being. What does a world answer when it is asked to explain all three? Ask one, for an agency that visits 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
}
```

✓ ran: printed what this block promises

It answered under three headings, `BEING`, `BECOMING` and `MET FROM WITHOUT`. `SetBody` only declares. No file was written by this block.

A world strives. A goal is a wanted state of the graph, and the gap is measured. A goal with no pattern to evaluate is refused, and never faked:

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'.
}
```

✓ ran: printed what this block promises

`Pursue` prints its own first line, and then the `?` prints the list it returns. One relation later the gap is closed: `0 proposal(s)`, and `GoalSatisfied` answers `1`. The vague goal is refused with its reason: `no 'every :Type' clause in 'make things nice'`. With a reaction declared, the same gap proposes a *visit* for Bilal, and without one, as in the run above, the default act is *attend*: `[ "attend", "client", "for", "bilal" ]`.

A CONSTELLATION

Nothing crosses between worlds unless it is declared and governed. A bond must exist, and then three gates stand on the crossing: the act must have a declared risk tier, the caller must hold the permission, and the caller's authority must cover the risk. Each refusal names its gate. Which gate refuses first? The bond exists in all four calls below, so the three refusals are the governance's:

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" ]
```

✓ ran: printed what this block promises

The risk tier refuses first: `'order-produce' has NO declared risk tier`. The doubled word "refused: refused:" is the library's own text. The cleared crossing wrote a decision by itself! Its record holds the authority and the risk of that moment, and a clock read, so the block prints only the rationale: `cross-world call cleared`. That is the lineage chapter 30 uses.

The shared ground is a world too, in the design's words: world zero, uniform and recursive, and not a special kind of node. Inside a constellation the commons is itself a platform, and it holds an identity, a session and a store. What class is the commons in code? Ask it:

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()
```

✓ ran: printed what this block promises

This block uses an in-memory database, so no file and no network. The session token is random, and only its presence is printed. Secrets are stored salted and hashed, with the engine's key-derivation function. The first line answers `stzplatform`, and that is the point: a platform is not only the envelope around a solution. It is what a constellation stands on. In code the commons is not a node of the constellation graph: when I probed it the graph held two nodes for two registered worlds. World zero is a role the platform plays.

Does a world swapped live keep its bonds? A constellation also nests in another, and a name is taken once:

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.
```

✓ ran: printed what this block promises

The bond survived the swap: `CallAcross` still answers `1`! Chapter 28 uses `Retire` and `Revive`; this block is `Swap` and `AddConstellation`. Nesting is a fact of the registry. Nothing in the code routes a call into a nested constellation, so the recursion of the design is a registry's recursion and not yet a call's. A crossing is a verdict: `CallAcross` clears or refuses with `1` or `0`, and nothing is handed to the target yet.

One trap. A registered world is a copy. The host language copies an object by value when it is stored, so the world you register is not the world you hold. The header of the class says it: world objects are reached through the registry index, the live path. Configure the world fully, register it, and reach it through the constellation from then on. The envelope has the same doctrine: setting a world on the platform harvests its declarations at call time, and setting the governance snapshots the regime.

THE ENVELOPE, AND THE SEAM

A platform grants a capability of a device only if governance clears its risk. Is a held permission enough? The risk tiers, two for the camera and four for payments, are the library's declared ones:

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)
}
```

✓ ran: printed what this block promises

The permission was held for both, and the payments were refused anyway! The reason: `'resto' holds 'delegated' authority (level 2) but 'use-payments' is risk tier 4`.

The class header says what the world lacks: the world models the world, and deliberately lacks the operational envelope. The envelope's duties are generation, the capability seam, the commons runtime, a networked body and a registry with enforcement. Generation is real, and refuses to generate for a world that declares no reach. The success path writes real files, so I do not run it here. I ran it once, for a world of one screen and one thing, and removed what it wrote. The web shell is a page of 318 bytes with the world's screens and things as list items, and the desktop shell a launcher of 251 bytes. Other worlds give other sizes. That is the limit of "per-platform shells": generation is real, and tiny.

THE ARCHITECT, AND THE THREE SCOPES

By *scope* I mean the System Foundation's: development, deployment, and runtime-current. Every system act is written in a named scope. A Softanza architect models the deployment architecture first. The same verb, in two scopes, gets two verdicts, and the target's own words say why:

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.
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The development system's name is the declared one, so it does not depend on the machine. The line `native` is host-dependent: it holds only on a host that can spawn a process, as chapter 31 says of its own word. The Android refusal, `scope 'android' (android) forbids 'process'`, and the capability list of the board are the target's, whatever the host. The three scopes are `new stzDevSystem()` or `DevelopmentSystem()` for development, which is tooling only; `App(:x).System()` for deployment, where feature code is written and checked; and `new stzCurrentSystem()` or `CurrentSystem()` for runtime-current, which is what runs now. The scope object has no `Capabilities()` of its own: the call goes through `.System().System()`. A request.

The architect's artefact is a text file. Can it be read back, and what does the platform say of a solution that is not sound? Run it:

```
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
```

✓ ran: printed what this block promises

The file form is plain text that the architect owns, and it round-trips from a string. The unsound one is refused before building, with both faults named: `the solution is not sound -- no development system declared (use DevelopedOn); the solution deploys no apps`. The order of building and deploying is chapter 33's, and the gate on who deploys is chapter 24's.

Who is the architect? The record names the architect twice: in the System Foundation, and in a memo of the estate for a school that would certify architects of Softanza, with three levels, a memo marked withdrawn on the day it was written. That certification is a proposal of 30 September 2026, not a delivery, and the site must not announce it. The estate's protocol defines a role as a profile held

in a scope. Applied here, which is my application and not the record's, the architect is not a rank. It is what one holds in the scope of a solution: the profile, the worlds, the governance. I hold this as a view, not a result. The site, as I read it on 9 October 2026, has a journey page for the platform architect and a certification page that says the proposal is not announced. I tell the architect as what one declares, and the certification as that proposal; no school is named here.

Seven refusals stand behind this form.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused the host and the mini-app. Worlds, not mini-apps; a governed graph, not a host container. Chapter 4 told it, and I point back.

I refused to make the commons a special kind of node. In the design it is a world, world zero, uniform and recursive; in code, as section 3 printed, it is a platform object and not a node of the graph.

I refused the sandbox as the definition of isolation. Isolation is a declared, inspectable meaning and not a black-box virtual machine. A world's sovereign body plus norm-gated bonds. A runtime sandbox is only an implementation for untrusted worlds.

I refused to make governance one more facet, beside the design's constellation, bonds, commons, reach and presence. It is the ambient. In code, one governance per constellation, configured through it, so that there is one live governance and never a stale copy held by a caller.

I refused to give the world the envelope. `stzApp` models the world, and deliberately lacks the operational envelope.

I refused to freeze *current* under a better name. The cure is not a better name: every system act is written in a named scope. The attempts it refuses are a pair of words, this system and the target system, and a triple, host, runtime and target.

I refused to deploy a profile. We do not deploy a profile; we deploy a platform and its parts. The profile only describes. The platform owns it and drives the building and then the deployment.

The court now says which of it runs.

5. THE COURT

`06_stzsuperapp_narrated` proves that worlds register as nodes; that the commons holds an identity and a store; that a crossing is refused for an undeclared risk, accepted when governed, and refused when unbonded with the bond named; that a retired world breaks its bonds and revives; that a swap keeps the node and the bond; and that a constellation nests in another. `01_platform_narrated` proves that shells are generated per declared reach and a world without a reach is refused; that a capability is granted only when governance covers its risk tier, and a capability never requested is refused with a reason. It proves that the commons registers identities, opens sessions, posts and reads messages, and stores; that the world is served over a real port; and that the registry refuses a call to an unbonded or retired world. It writes shells beside itself and opens a loopback port. `full_scope_model_narrated`, which the guard calls the architect's common ground, proves that the solution is modelled first, that an unsound one is flagged, that the three scopes are distinct, that the scope refuses what the target forbids and rehearses what the host lacks, that the text file round-trips, that building precedes deploying, and that a deploy by a language-model actor is refused. `04_stzapp_pursue_narrated` and `05_stzapp_live_narrated` carry the goal and the reactions.

The guard the chapter does not have: none judges a world's truths, a flow, or the time of a reaction.

Next I set the record beside the field.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The super-app leads in users. The design names its model and replaces it; chapter 4 says the record measures nothing against them. The Platforms page prints the other half: the customer platforms run today on ordinary web stacks. A customer platform on the engine, as a constellation, is the direction; chapter 33 carries it as the next line.

The field leads in shells. The design promises a progressive web app, a desktop shell, a mobile shell. The code generates a page and a launcher of a few hundred bytes. The shells are designed. Generation is built, and tiny.

The field leads in what a bounded context carries: a team and a language. A world carries a graph and a narration. The persona a world is for is not a field of the class: a world has the list of surfaces it reaches and nothing else about whom it serves. The site's five things, a shared ground, a world for each person and device, bonds, rules, roles, are true of the commons, the bonds and the governance. The roles are the actors, permissions and authority of the governance. There is no role class, and a world per person and device is a naming convention.

The Atlas does not rate this plane. `stzApp` is named once, in the lane of the area for interfaces called App shell and windowing. That lane, not the world, is rated solid, with the note that there is no multi-window orchestration, dialogs, menus or tray. The platform in a lane of the area for the system. The constellation in none.

The compiler leads on scopes. It checks against a target in daily use at scale; Softanza's scope is a runtime object that a program asks. What it adds is a verdict at the call site in the target's own words, and a twin that rehearses what the host lacks, which is chapter 24.

Where the field and the record meet: a super-app is a graph of graphs. The field's context map is a diagram. Softanza's is a registry that refuses.

7. THE LIMIT

BUILT	a world as a declaration with a being, a becoming and a way of being met; a goal whose gap is measured; a constellation of worlds with bonds and three gates on every crossing; the commons as a platform; the capability seam; the profile of a solution as a text file, and the three system scopes
DESIGNED	the near-natural idiom of the constellation; a call routed into a nested constellation; shells that embed the engine; the architect's certification, which is a proposal; the five-row guarantee sheet, as a requirement
ABSENT	a world that judges its own truths; a narration for the constellation or the platform envelope; a written five-row guarantee sheet for any solution

A world's truths are narrated and not judged. The number in a reaction is the first case. Do ninety days and one day give different proposals? A reaction fires for each instance that lacks a relation named after the thing it proposes, and the comment above `Pulse()` says so. Run two clinics:

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" ] ]
```

✓ ran: printed what this block promises

Both clinics answer `[ "propose", "checkup", "for", "bob" ]`. Ninety days and one day give the same proposals!

The same reading holds for `IsTrue`, which is stored and narrated while instances carry no fields to judge it against. It holds for a flow's `Require` and `Then`, stored and narrated, where I found no executor; and for the bounds of a refinement. A world today declares that a balance must not be negative. It does not judge a

balance. What it does judge is a goal's gap, a crossing and a capability. So I say "a world that wants" for the goal, and "narrated" for the truth. Chapter 4's `true when balance @ >= 0` is a declaration only. So the chapter says it in one line: declared and narrated; judged only for goals, crossings and capabilities.

The design writes more than the class speaks. Does the class speak the design's near-natural idiom for the constellation? Ask it for `Hosts`, `Admits` and `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
```

✓ ran: printed what this block promises

Three times the answer is `Calling Method without definition`. `Shares`, `Provides`, `Norm`, `Reaches` and `Generate` are not defined on a constellation either. The class has `AddWorld`, `AddConstellation`, `Bond`, `Retire`, `Revive`, `Swap`, the `Gov*` family, `CallAcross`, `Crossings`, the commons calls, and calls that look: `Why`, `GraphQ`, `WorldQ`, `KindOf`, `IsActive`, `HasWorld`, `NumberOfWorlds`, `AreBonded`, `CrossingCount` and `LastCallMs`. The verbs of composition are these. The design's three aspects are reached today by `AddWorld`, `Bond` and `OpenCommonsOn`, and governance by `Gov*`; reach and presence are the part that is not there. The design's own header calls the document a forward design.

What is built: the world; the constellation with its gates, its ledger of crossings with timings, a decision written on a cleared crossing, and a refused crossing noted as evidence; the platform's capability seam, its commons, its registry, its shells, its profile with building then deploying; the three scopes.

What is designed: the near-natural idiom; a call routed into a nested constellation (today a crossing is a verdict, and nothing is handed to the target); loading one world without rebuilding a host; shells that embed the engine; the five-row guarantee sheet, which the vision asks for and the site's journey page marks designed: one row for each of five systems the solution answers to, technical, social, ecological, economic and cultural.

What is absent: a narration for the constellation or the envelope, since none exists. The narration for building a first app uses verbs the class does not have. Seven of its calls fail at this commit, and I cite it for the idea and not the syntax. A customer platform on the engine as a constellation. The written five-row guarantee sheet: no solution has one. A lane of the Atlas for the world, the constellation or the envelope.

The six doors come next.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You declare a world with braces, ask it to explain itself, and cross between worlds with a call that returns 0 and a `Why()` that names the gate. Watch the copy: configure, then register.
The analyst	You declare things, relations and a goal as a wanted state of the graph, and the gap is measured. Your truths and durations are narrated and not yet judged.
The designer	A world declares screens as intents, a refinement as a bounded knob, and where it appears. The shells generated today are stubs, and the persona a world serves is not a field.
The decision-maker	You declare who may cross where and at what risk, in three gates, and every cleared crossing leaves a decision: the block printed its rationale, and its record holds the authority and the risk of its moment. Your architect's file is text you own.
The educator	The record is thin. The education plan lists a ladder from practitioner to architect to contributor among five competing level schemes, and I say so. The exercise below is judged by running.
The agent	A world can be wrapped as an AI agent that the host ticks. A deploy by a language-model actor is refused, and nothing is deployed. The crossing between worlds is governed by the calling world's name and authority.

9. EXERCISE

Declare two worlds for a place you know: a shop and its supplier, a class and its library, a clinic and its pharmacy. Give each a thing. Join them with one bond for one act. Before you run anything, write what you expect `CallAcross` to return, and which reason you expect to read first.

HARO

```
? YourConstellation.CallAcross("a", "b", "act")
#--> 1
```

✎ not run: an exercise, written for you to run

Then ask for the call, read the reason, and repair the declaration, not the court, one gate at a time, until it returns 1. Count the refusals. Which gate would you have forgotten without the court? The court tells you which gate refuses: a missing risk tier, a missing permission, an authority that does not cover the risk. It does not write the declaration. A harder one: retire the second world and predict the new reason before asking; then register a copy of the first, change your original, and predict what the constellation sees.

I left this one in your hands. Take the gates block of section 3 and delete one line: `oCon.GovSetAuthority("resto", :Delegated)`. Stop the block at its last `Why()`, as here, and I print nothing under it:

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()
```

✎ not run: an exercise, written for you to run

Run it and read the last `Why()`. Name the gate that refuses, and say what the block would have to say for the crossing to clear. Section 3 says what each of the three gates asks, and that each refusal names its gate. When you have solved it, you will know that the court names the gate that refuses and does not write the declaration.

The next chapter is about how a declared solution reaches reality, through exactly one door: rehearse on what touches nothing, commit through a governed actor, and let production refuse anything that is fake. Rehearse, then commit.

WHERE THE EVIDENCE LIVES

NARRATION	`stzApp-building-your-first-app-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
NARRATION	`stz-built-on-real-world-need.md`
ON THE SITE	platforms
ON THE SITE	offering
ON THE SITE	forged
RULINGS OF THE STUDY	12.2, 12.12, 12.13, 12.15, 12.19, 12.25, 12.29

PART V · CHAPTER 24

Rehearse, then commit

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 11 of 11 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the three practices the field keeps beside the program, plan and then apply, the test double and the emulator, and I show the one path the library puts in their place. I rehearse on a twin of files and a twin of the process environment, and I read the plan of placement. I build the emulator's bundle, run a deployment that fails and rolls back, and code against services by name before I subscribe to any. Then I ask production to refuse, and I print the one fake it does not refuse: a double that says nothing about itself. The principle is joined from three documents of the record. Reality is reached through exactly one door: rehearse on what touches nothing, commit through a governed actor, and let production refuse anything that is fake.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Three practices of the field stand beside the program, each with its own configuration.

The first is plan and then apply: infrastructure as code and deployment pipelines. The record's own references are the dependency graph of a pipeline's jobs, a tool's resource graph with apply-or-revert, and a cluster scheduler that places work by the resources it asks for. The field's habit is that the plan is a file for a tool, and whether the wrong actor may apply it is a policy. The record states the loss it answers: it's a convention, enforced by hope.

The second is service virtualization and the test double: a vendor's test mode with published test cards, a stub server for HTTP, and the libraries that record a call once and replay it. The field's habit is that the double lives in the test suite, and production is a different file. The record's complaint: no gateway will decline a card because you asked nicely.

The third is the emulator and the simulator: a phone emulator, a board emulator, a local stand-in for a cloud. The record's own sentence on the genre is that simulators lie. They mock the database, stub the device, approximate the compute.

What the three share is that each is a separate tool, and the program itself knows nothing of it. That is general knowledge of the tools, and not a study of the record, except where I quote the documents.

2. THE FIRST PRINCIPLE

Reality is reached through exactly one door: rehearse on what touches nothing, commit through a governed actor, and let production refuse anything that is fake.

The record's first principle for this ground is the first clause, and the library's own label calls it a first principle: reality is reached through exactly one door. The default path for consequential change is to rehearse in a twin, generate a plan, and commit the plan. The plan is the door. The framework's summary is three words: rehearse, plan, commit. The structure line of this book's outline says the last clause: production refuses a fake. The first clause is a library document's, the System Foundation's, written with no author line. The middle clause, commit through a governed actor, is the deployment design's: "only an effectful actor commits". The chapter joins the three and attributes each part to its source.

3. THE SOFTANZA FORM

THE TWIN

The one artefact that crosses toward reality is a plan. It names each step. It flags the one that removes something. And asked to validate, it re-reads reality. Chapter 20 said "I saw it warn" and did not show the words. What does the plan say about the source it was going to move? Here are the words, and nothing is executed:

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" ] ]
}
```

✓ ran: printed what this block promises

The plan answers `source absent in reality: old_settings.txt`, and it flags the move: `MOVES (removes source)`. The library's narration prints a closing line, that reality is untouched until I choose to commit this plan. The run does not print it, so I do not show it.

The twin is not only a twin of files. A variable set, a directory changed and a child process queued are recorded, narrated and flagged, and no process started. Does the real environment have the variable? Ask it, after the twin has set it:

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" ]
]
```

✓ ran: printed what this block promises

The twin holds `rehearsal` and one pending spawn, and the real environment answers `0`! Chapter 20 counts both twins as built, and this is the second one.

The twin has a second job. A programmer on a laptop with no pins writes pin code for a board. The scope says *rehearsed*, as chapter 8 showed and chapter 31 shows. At deploy, the lowering bridge turns the rehearsal into the source of firmware. What happens when an assistant asks to deploy? `Build()` records the build command and does not run the toolchain, and `Deploy` lowers in memory, so no file is written:

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);
        #--> }
}
```

✓ ran: printed what this block promises

The assistant is refused, and told that it `may propose a deploy, not perform it`! The person's deploy yields the firmware. The limit of this block is in section 7: only two verbs are lowered.

The twin rehearses; next, the plan says where each part would go.

THE PLAN OF PLACEMENT

Before a byte is built, the delivery planner states, for each capability a part needs, whether the target does it, the engine does it, a construct does it, or the server does it. It gives the reason in a sentence. The figure in parentheses is the catalogue's estimate and not a measure, so do not quote it as a claim. "1 parts" is the library's own grammar:

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.
}
```

✓ ran: printed what this block promises

The rule behind the plan, in the emulation design: the edge carries a capability only if it is critical, and either unique to Softanza or weak or absent on the target platform. A regular expression is a construct over the platform's own, and never the engine's matcher compiled for the browser.

The plan says where; the emulator builds what the plan says.

THE EMULATION

The emulator turns the plan into a bundle, and the bundle is what ships. In the next block the bundle generated for the rehearsal is wired straight into the production attach, and the actor decides. What does the site become when an assistant calls `Deploy(:Production)`, and what when a person does? The manifest that lands names the engine group the part carries. The target here is a local folder, the kind of site that needs no tool but file operations. The block writes two folders in the working directory, `_bundle` and `_live`, and removes them at its end:

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")
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The assistant left the site `absent`, and the person made it `launched`. The honest half: the manifest's artefacts line reads `index.html,stz.js`, and no engine file for the web part is in it, because the engine compile was off. The block says what the run shipped, and not what a full run would. A bare `Deploy(:Production)` with no actor at all returns a rehearsal and writes a warning, "no effectful actor --

returning a rehearsal (nothing committed)". I ran it too, in my research run of 8 October 2026 and not in a block shown here: the site stayed absent and nothing was written.

I did not run `Deploy(:Emulated)`, because it builds each part's subset through the engine toolchain, and the toolchain's build is something this machine must not run. What I ran is the bundle without the compile. In my research run of 8 October 2026, outside this chapter's blocks, its map named an engine file that is not there. Section 7 says what that means for the claim that the engine runs in a browser.

THE DEPLOYMENT

For a local site, a rollout that fails undoes what it had done; the other kinds of site only try, as a best effort. The web part deploys and passes its gate. The api part's store fails. The remaining steps are skipped and the deployment does not commit. What becomes of the web site, already live? The failing site is a server with no storage location, which fails without spawning anything. The block writes one folder, `_site_cdn`, and removes it:

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")
```

✓ ran: printed what this block promises

The api store reads `FAILED`. The web site, whose three steps read `done`, reads `rolledback`! With an assistant as the actor, the same run returns every step as *rehearsed*, the site stays absent, and no folder is created. I ran it in my research run of 8 October 2026, outside this chapter's blocks, and do not print it.

The deployment fails safely; the services come next.

CODE FIRST, SUBSCRIBE LATER

The code asks for a service by name and never learns which implementation it got. The sandbox sees the first call and none after I rebind the service as live; the second call goes to the live adapter. The surface is sound once every live binding names a secret that the store holds. The live gateway in this block is a stand-in class written here, since the library ships no live card-gateway adapter. A stand-in named `LiveGateway` that returns "authorized" is a fake in everything but its name. The registry takes it as live because it declares nothing, and section 7 says why that matters:

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
```

✓ ran: printed what this block promises

The day you do subscribe, you change a binding and not a line. The key is an invented test string and not a real one.

PRODUCTION REFUSES

One report names five of the eight ways a surface is not fit for production. Four are errors: a declared service never bound (`unbound-service`), a fake (`sandbox-in-production`), a real database that vanishes on restart (`ephemeral-in-production`), and a live adapter whose key is not in the store (`live-without-secret`). The fifth is a warning: a live adapter that names no secret at all (`inline-credential`). Chapter 31 shows the unbound one and the raise. This is the rest of

the list. In the development phase the surface is already not sound, since an unbound service and a missing key are errors in any phase. The other two errors appear only in the production phase. The code lists eight invariants, and the service plan (section 4) and the narration list five:

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
}
```

✓ ran: printed what this block promises

Two different questions are asked at the door: who is refused, and what is refused. Why is each refused? Ask the registry twice, once with an assistant and once with a person, in the development phase:

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
}
```

✓ ran: printed what this block promises

The assistant is refused for who it is: `actor 'assistant' is not effectful -- it may propose, not commit`. The person is refused for what the surface is: `sandbox-in-production: still bound to a SANDBOX in a production phase -- a fake must never ship`. And asking leaves the phase where it was: `development` before, `development` after, though the question is asked in the production frame.

And the registry knows services while the delivery knows parts. Joined, they answer a question neither can alone. Which part is about to ship on a double? The registry itself reports nothing, since the phase is development and there are no findings. The part-level check answers anyway, by destination, and names the part:

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]
```

✓ ran: printed what this block promises

It names `api` twice, with `[error]`, and `web` once, with `[warning]`, where the registry reported nothing!

That is what the run showed; the decisions behind it come next.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

Everything below is from the library's design documents and narrations, written in sessions with no author line. The record holds no ruling of my own for this chapter. Its decisions are the documents'.

I refused the simulator that lies. The emulator does not mock: in the design's words, each part runs the same compiled engine artefact that will ship to its target. That claim has a caution, in section 7: the proof that the engine runs in a browser is not in the pinned tree.

I refused to re-ship what the target does well. The edge carries a capability only if it is critical and either unique or weak on the target.

I refused tool sprawl, and the convention enforced by hope. The target should be one config away, and deploying to it should be one governed act.

I refused an inline secret. A site carries an auth reference, not a key.

I refused to touch reality by default. Production commits only if the actor may commit. A bare `Deploy(:Production)` with no actor touches nothing.

I refused the agreeable fake. An agreeable fake is more dangerous than no fake at all. I hold this as a view, not a result. Silence is the worst answer: the lenient mode answers 501, because a miss must never be mistakable for success.

I refused "fake versus real" as the whole truth. It was too coarse. There are four postures now: sandbox, local, live, and conformance. The local posture was added when the database exemplar showed that a SQLite file is a real database in a file you own.

I refused a gate that fails open. The registry once kept its state in attributes. Attaching it to a delivery took a snapshot, so the gate would have passed an unsound surface. A gate that fails open is worse than no gate. The state now sits in

a table keyed by identity, so every copy is the registry. Anything that governs others must survive being copied.

I refused to virtualize the quality of an AI model. Frontier-model quality cannot be virtualized. What the sandbox sells instead is determinism, a cost you can assert, and working offline.

I refused to ship the live adapters. The service plan says the live Stripe, PayPal and Adyen adapter is deliberately not shipped, since an account, a key and a network are infrastructure-gated. A hub adapter is built, and chapter 31 shows it. The contract is defined, and the registry refuses to let the fake ship in its place.

I refused to ask a question in the wrong frame. `MayGoLive` once asked about soundness in the current phase, so with a fake bound it answered yes. Now it is asked in the production frame whatever the phase.

I refused to say the restaurant's need was the design's origin. The site says that the emulation and deployment designs take a solution as an example. It is an example, not the origin of the design. The sample solution in the documents carries a customer's product name, and I have renamed it in every block. Chapter 28 tells the story, and I do not repeat it here.

The service plan is headed "written 2026-07-23 in answer to the user's question". The question asks whether emulation covers databases, business APIs, frontier models and cloud providers. The aim is that a programmer can code a whole program without subscriptions or fees and still get a working solution ready to deploy. The chapter tells the plan's own heading and date. The record does not say who asked, and I do not claim it was mine.

5. THE COURT

`service_registry_narrated` proves the declared surface, that an unbound service raises, and that the application asks by name. It proves that a sandbox in production is a violation, with one finding per fake, and that the credential must

be in the store. It proves that going live is a governed crossing, with an assistant refused and a person refused over a fake, and that asking leaves the phase. It also proves the CI helpers, the fourth posture, and a live adapter that needs a certificate. `service_delivery_narrated` proves the external surface rehearsed in the plan, and the pre-flight. It proves the gate, which in a production deploy refuses a fake and logs every reason, and opens once the surface is real. It proves that the registry is one registry however many copies are made, and the question of which part depends on a fake. My research run of 8 October 2026 ran both: 93 of 93 and 60 of 60. The second left a file of zero bytes beside itself, and I removed it.

`virtual_system_twin_narrated` proves, in twelve scenes, that the twin touches nothing real, that the plan narrates and flags, that commit is the one door, that a scope refuses, and that a reviewer rejects a step. It proves, too, that a hundred operations stay in memory until one commit. `virtual_environment_twin_narrated` proves the same shape for variables, a directory and queued spawns. `emulator_narrated` proves that the bundle generates, the map renders every part by role, and each part carries only its subset. `deployment_narrated` proves the site as config, the governed commit, the step plan, the rollback, feasibility, and the git backend; it spawns `git`, so the book does not run it. `full_scope_model_narrated`, scenes 13 to 16, proves that building precedes deploying, that an unsound solution cannot be built, the lowering bridge, and the governed deploy.

The refusal that production makes is proved by the first two guards, which ran. That the rehearsed engine runs in a browser is the next proof: no guard that runs checks it today.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

Real services lead. The plan says it: a double is not the real service, and you still need at least one real run before production. A scripted gateway is not a vendor's edge cases. Recorded answers drift, and a replay suite needs re-recording and contract tests against the real API. The library has the recording call; a drift check for recorded answers is designed.

Frontier AI models lead, and cannot be emulated. The AI model is judged against the real one, deliberately, as its own activity.

Live infrastructure leads. The deployment is infrastructure-gated: live verification against a real host or account. The commands are generated, and one reachable host away from literal.

The mature deploy tools lead. The narration says it plainly: this does not replace every mature deploy tool, it unifies the seams those tools leave exposed.

Fidelity has a stated boundary. Emulation is faithful but not omniscient, so it says which is which. What a double proves is narrower still: a capture sink proves what you would have sent, and a directory is not a CDN.

Where I think the record stands apart is the coupling. The double, the plan, the credential names, the gate and the actor are declared in the same program that ships, and one gate judges them.

That is the field; the limit is where I stop.

7. THE LIMIT

BUILT — a twin of files and a twin of the process environment, each with its history, snapshots, a narrated plan, its risks and, for the file twin, a validation against reality; the placement plan; the emulator's bundle, wired into the production attach; a deployment as a graph of steps with a rollback; the registry of services with its postures and the gate that refuses a fake at production

DESIGNED — the engine seen running in a browser; a drift check for recorded answers; branches of a twin; twins for databases, interfaces and networks; the emulated phase set by the emulated deploy

ABSENT — any live card-gateway adapter for payments (a hub adapter is built and proved only against a twin, chapter 31), an AI model, mail, SMS or an object store; live verification of a remote host; a person's look at the emulator's surface; a customer solution emulated and deployed through this path

A hand-written fake that says nothing about itself passes the gate. The registry reads what the object says it is. A double that declares neither that it is a sandbox nor that it is local and real is taken for live: no opinion means live. So "production refuses a fake" is exact for the library's own doubles, and for any double that declares itself. What posture does the registry give a double that declares nothing, and is the surface sound? Ask it in the production phase:

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" ]
```

✓ ran: printed what this block promises

It answers `live`, and `IsSound()` answers `1`: the always-yes double passed! The stand-in of the earlier block, `LiveGateway`, is such an object.

The lowering knows two verbs. Only `read_pin` and `write_pin` become code. Any other rehearsed operation becomes a comment line. In my research run of 8 October 2026, outside this chapter's blocks, I rehearsed `UseIn(:sensor, :gpio, "blink the status led")` and the firmware got a comment. For a server or a phone, the artefact is a header with no lines when nothing was rehearsed. The narration's "the board is the last inch" holds for reads and writes of pins only.

The fidelity contract is by target class. The design says each part declares what is faithful and what is approximated. The code returns one fixed text for a microcontroller, listing two approximated things, and one for everything else,

"faithful".

The sizes are estimates. The plan prints a figure from a catalogue, and the narration prints two different figures for the same part. Do not print a size as a claim.

`Deploy(:Emulated)` does not yet move the registry. The plan says it resolves services to sandboxes. The code builds the emulator and returns; only the production deploy touches the registry's phase. The `:emulated` phase value that the registry has is set today only by a guard; setting it from the emulated deploy is designed.

A refused `Deploy(:Production)` leaves the registry in the production phase. After the refused deploy of my research run of 8 October 2026, outside this chapter's blocks, `Phase()` printed `production`. The pre-flight restores the phase, and the deploy does not.

A plan flags today; ranking is the direction. Chapter 20 said it, and the framework document says "ranked risk assessment".

What is designed: the engine running in a browser. The documents say that the phone's `solve 2 -8` returns `x = 4` in a page. The guard that checks it, `wasm_narrated`, says in its own header that it checks the wiring and the build, since the host language cannot execute wasm. It cites an in-browser page with "8/8 checks". That page is not in the pinned tree: I searched for it and found none. And I have not seen the emulator's surface in a browser, and the record names no one who has. By the law of chapter 33, that surface is unperceived. So the chapter says: wired, its build checked by a guard that I did not run, not seen in a browser by anyone the record names.

Also designed: branches of a twin and the plan's alternatives and feedback, for which no method is built yet; twins for databases, interfaces and networks; and a drift check for recorded answers.

What is absent: a live card-gateway adapter for payments (a hub adapter is built and proved only against a twin, chapter 31), a frontier model, an object store, SMS or mail. The contracts and doubles ship, and the live clients do not, deliberately. Live verification of a remote host or account: commands are generated, and nothing ran against a host. A customer solution emulated and deployed through this path: nothing in the record yet, which chapter 33 carries as the next line.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You code the whole integration against names, with doubles that raise when you forgot something and a counter that lets you assert a fee. The day you subscribe, you change a binding and not a line. The loss: a double you wrote yourself that declares nothing is taken for live by the gate.
The analyst	Before anything moves you can read the placement plan, the external dependencies with the credentials production will ask for by name, the step graph, and which part depends on a fake. The sizes are estimates, and the findings are by part.
The designer	The emulator generates a surface from the plan, with a parts grid by role and one window per part. I checked the generated text only, and never the look. Say "generated", and not "seen".
The decision-maker	"Who may deploy?" is a property. An assistant may plan the release and the site stays absent. A person may commit. Neither may ship a declared fake. The loss to print: live hosts and real providers are gated by infrastructure, and the record names no live run.
The educator	A learner can build a whole shop's integration with no account, no key and no fee, and watch the double refuse what she forgot. Her deployment fails safely and rolls back.
The agent	You may rehearse the whole integration and plan the whole release, in a world where nothing real can move: a sandbox payment moves no money and a mail sink sends no mail. You may ask whether you may go live and be told why not. You cannot commit. The loss: that guarantee covers the library's own doubles, and a surface bound to something live is outside it.

9. EXERCISE

Write, under each line, what you expect before you run it:

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()
```

✎ not run: an exercise, written for you to run

Then register a key named `maps_key` in the store, and write the number of production findings again. Then bind a hand-written class with an `Authorize` method that always says yes, and ask what posture the registry gives it. If a promise breaks, do not change it: explain the output first. I do not tell you where you will be surprised.

I left this one in your hands. The block below is the one from section 3, "Code first, subscribe later", with one name changed. I changed it on purpose:

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
```

✎ not run: an exercise, written for you to run

Run it. Compare the answer of its last line with the one section 3 gave, and explain any difference before you touch the name; section 3, "Production refuses", shows how to ask the registry for its report. When you have solved it, you will know what makes a live binding sound: a secret that the store holds, and not only a name.

The next chapter turns from the services behind a program to its users: your users keep their medium, and a new medium adds renderings, never verbs.

WHERE THE EVIDENCE LIVES

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
ON THE SITE	forged
RULINGS OF THE STUDY	12.13, 12.19, 12.21, 12.22, 12.25, 12.28, 12.29

PART V · CHAPTER 25

Your users keep their medium

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 11 of 11 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take one sentence of the interface constitution, a new medium adds renderings, never verbs, and I run it over the medium I can measure. A question becomes a line of text, goes out over an SMS sandbox, and is billed per segment in two alphabets. I show what the library ships, the question as data and the court, and what you write, the rendering and the reader. I report the record's refusals as the record's: a conversation that needs no AI model, and a voice that hears commands and not dictation. And I print the limit whole: no messaging app, no USSD, no telephone line, and a reply with a space in it, which the conversation cannot yet take.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Conversational commerce, or omnichannel: a business reaches its customers in the channels they already use, and a dialogue engine sits behind them. These are the practices, from general knowledge and not from a study of the record.

The dialogue frameworks own the dialogue model. The designer writes intents and flows in the framework's notation, and the framework understands free text with trained AI models and ships connectors to channels.

The business platform of a messaging app: messages arrive by a webhook, and the business may answer freely inside a service window that the customer opened. Outside it, only approved templates, through a verified business account.

The operators' gateways: SMS and USSD. SMS is billed per segment, and I measure this myself below. USSD is a session of short screens with a timeout, through an operator's gateway, country by country, on any phone without data. No file of the snapshot says anything about USSD.

And platform speech, in the library's own plan: Windows has one speech interface, macOS has another, and every modern browser has a third.

What they share is that the dialogue is written once per channel or once per framework, and the channel is a connector, a deployment detail. The chapter turns that picture around. The world is declared once, and the medium is a rendering of it.

2. THE FIRST PRINCIPLE

A new medium adds renderings, never verbs.

It is a sentence of the interface constitution, printed on the site: "The grammar is closed: a new medium adds renderings, never verbs." The record is silent on whether the wording is mine. I quote it as the constitution's. I do not tell it as my own sentence. The twenty-two verbs it speaks of are the constitution's, in six families. Appendix C prints them as I read them from the constitution's own file on 9 October 2026. The repository is unnamed, since it is not public yet.

The same sentence, built, for one family of values and not for the twenty-two verbs: the value is shared, and how it is rendered is each medium's own. It is a ruling by Central, the estate's coordination entity, on my delegation, of 30 September 2026. It proves that the principle is not only a slogan. It is the shape of three working faces: colour, earcon, phrase.

For the floor of the dialogue, the code's own sentence: deterministic floor, no model needed. Chapter 17's law, applied to a medium.

3. THE SOFTANZA FORM

The library ships the question as data: a force, a subject, a relation, options and a reason. It ships the verbs, ask, reply, refuse and conclude, and the court. A medium is a rendering of the data and a reader of the reply, both written by the crafter in a few lines. Nothing in the library renders a question to a phone. What it ships is an English line, `AskIn`, which is already a text menu: "(1) cheese (2) olives".

Two small functions of my own are the whole of the medium: one that renders the question to a line of text, and one that reads the reply. The reading matters, and I show why in the third block. What does one question cost, in Latin letters and in Arabic? Run it:

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")
```

✓ ran: printed what this block promises

The question is data, so any medium renders it. And the medium's own law, here that SMS is billed per segment and the alphabet decides, is a test and not a surprise. The English text is 93 characters by my count, and the block prints one segment. The same frame with two Arabic dish names is 89 units by my count (the two names are four letters shorter). The block prints `ucs2` and two segments, because the encoding changes. Four letters shorter, and one segment more! The telephone number is invented, and only the country code, which is Niger's, is real.

Now the whole conversation, rehearsed over the SMS sandbox before any account exists. A conversation is a loop of the library's verbs: the medium carries each question out and each reply in. The assertions are the bill and the refusal. What does the sandbox count, and what does the court say to "7"? Nothing in the loop is an AI model:

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" ]
```

✓ ran: printed what this block promises

Four messages, four segments, and the court's reason verbatim: `no option (7) was offered -- 2 option(s) on the table`. The refusal of "7" is the library's own, as in chapter 18; the difference is that it arrived from a phone.

The reader of the medium is the medium's job. An inbound SMS is a string. The library separates numbers, which are option indices, from strings, which are values, and it does not guess. Its own comment: an index can never be mistaken for a literal value. So the medium's reader must decide which it is. With no options on the table, what becomes of the text "2"? Look:

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" ]
```

✓ ran: printed what this block promises

A dish named 2! In the block before, my `ReadReply` turned the text "2" into the number 2, and the option was picked. This is the sharpest sentence about the bridge: a medium adds renderings, and it adds a reading too. The reading is where

a user's culture lives, the digits, the words, a local "OUI". I hold this as a view, not a result.

An outage is not a rejection. The medium refuses in its own terms. A malformed number, with no +, is not an E.164 number: fix the data. An outage is another matter: retry. What does the sandbox answer to each?

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
}
```

✓ ran: printed what this block promises

`rejected` for the number, `refused` for the outage, and nothing counted. The retry is `sent`, and the count is one.

An inbound message is believed only after its signature is verified. The way in is the pattern the payments plane built, and the verifier is generic, over the body. No provider's header or body format is modelled, and the JSON is invented. Times are passed in, so no clock is read. What does the verifier name when one character of the body has changed since it was signed?

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)
}
```

✓ ran: printed what this block promises

The first body gave `provider`. The altered one gave `[]`, and the reason is `signature mismatch (tampered body, or a secret this receiver does not hold)`. One character changed, and no provider was named! A replayed body inside the window gave "replay detected" in my probe.

One value, many renderings. The bridge's shape, built for the five semantic values of the interface constitution. Colour, earcon and phrase render one value. `muted` is silence in sound and a treatment in colour. A value the law does not know is refused:

HARO

```
? @@(StzSemanticValues())
#--> [ "danger", "warning", "info", "success", "muted" ]
oE = new stzEarcons()
? oE.WouldFireAt("Catastrophe", 1)
#--> 0
? StzColorToNumber("danger.muted") != StzColorToNumber("success.muted")
#--> 1
```

✓ ran: printed what this block promises

The voice: spoken, heard, meant. Text to sound, sound to text, text to meaning. The medium called voice renders and reads with a closed grammar, and the meaning is my declaration. This block needs a French voice and a French recogniser. This machine speaks two languages, English and French, and hears one, French, and a machine without them would give another answer, so the

output shown is the one recorded on this machine. No microphone, no file, no network: the recogniser is the in-process one. What does it make of a phrase it was taught, and of one it was not?

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
}
```

✓ ran: printed what this block promises

The taught phrase gave `danger`, the phrase it was not taught gave no match (`1`), and `Catastrophe` was refused (`0`). The refusal's reason says that `catastrophe` is not one of the five semantic values. I do not print the recogniser's confidence, which varies by machine.

The world declares its verbs and its reach, and its code names no medium. A world holds screens declared by intent, five of the twenty-two verbs, and a list of what it reaches. Nothing refuses `:whatsapp` when it is declared:

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
}
```

✓ ran: printed what this block promises

`Acts(...)` is stored and not narrated: the basket's line reads `select dish`, and nothing of `:Confirm`. And the stage as a run: what does the platform say when it is asked to generate the surface the world declared? This block creates an empty folder in the working directory:

HARO

```
oPlat = StzPlatformQ("stall-platform")
oPlat.SetWorld(oApp)
try
        oPlat.Generate(:whatsapp)
catch
        ? cCatchError
done
#--> stzPlatform.Generate: unknown reach surface 'whatsapp' (web/desktop/mobile).
```

✓ ran: printed what this block promises

`unknown reach surface 'whatsapp' (web/desktop/mobile)`: the platform refused it! It names the three surfaces it has, and the same refusal answers `:ussd`.

What may be said to a customer is a closed set. The court of what is said has a mechanism today: a closed enumeration compiled to a grammar. Chapters 11 and 19 use it for a language and for an AI model's output. Here it holds the replies that a business is willing to send. It is not a messaging-template unit, and there is none. What does the court say to a sentence the business never approved? Two things to print with it: the message says "the model answered" whoever the speaker is, and the enumeration is case-folded:

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.'.
}
```

✓ ran: printed what this block promises

The approved sentence passed (`1`), the invented one was refused (`0`), and the court's reason begins `[oneof @ reply] field 'reply' is a closed enumeration`.

THE RESTAURANT'S MEDIUM

Chapter 1 says that the owner of the restaurant made his first demonstration alone, on his own phone, with a messaging app as his support line, and the record holds that: the checklist of the 15 August 2026 demonstration, and the measurements of 19 August that he sent over the same app. A support line is not a medium. Chapter 2 already gives his tool as a spreadsheet. His own report of 4 July 2026 gives his current tool as a spreadsheet. It asks for a web page opened by a QR code or by a button for a social network. Nothing heavy to download. The line about various apps, a messaging app among them, is the label of an option in the scoping form, and it is not what he ticked. "Simple like a messaging app" is simulated in the project, as chapter 28 already found.

So the restaurant is the first form of the sentence of this chapter, and not an example of the messaging medium. The customer keeps their medium: a link that opens in the browser they already have. The messaging medium, as a thesis, is unbuilt. I tell this story and claim no more of a messaging app than the record holds: a support line, and the photos of one day. Chapter 1 and this section say the same.

That is the form, built and run; next come what I decided, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

The refusals below, after the first, are the record's: a session's decisions, written in the headers of the library's sound files and in its voice plan. I report them as such, and they are not rulings of mine.

Messaging, the question. My ask, as my study of the record reports it: conversations happen programmatically by default, or through minimal configuration, without obliging the users to change their usability cultures. The proposals for the conversational surfaces that follow it are a session's, and the record holds no ruling of mine that adopts them. I say *proposed*, not *decided*.

A conversation needs no AI model. The floor is deterministic. A model can upgrade the fluency behind the same surface.

Voice is not a paradigm. The plan of record says out until asked in writing: all of dialogue management, wake words, intent parsing and barge-in.

Commands, not transcription: a measured refusal. The gate was set before the numbers: free dictation ships at eighty per cent exact and 0.60 mean confidence. The run gave 66.7 per cent and 0.600, which failed. The closed grammar gave a hundred per cent and 0.898, which passed. So this face has no `Transcribe()`, and never will until a better recogniser exists.

Refuse, never substitute. Speaking French to an operator who asked for English is worse than saying nothing.

No inferred meaning. The crafter still declares the meaning. A sixth value would be a constitutional amendment wearing a library's clothes.

A sink proves what you would have sent. The SMS port's honest limit: carrier filtering, sender-ID rules, per-country alphabet quirks, and asynchronous delivery receipts. None of it is modelled here.

The screen acknowledges, and the sound is not the acknowledgement.

The next section names the guards that stand behind these runs.

5. THE COURT

`sms_port_narrated` proves the segment law, with its boundaries at 160 and 161, 306 and 307, the emoji cliff at 100 to 102 units, 71 Arabic letters making two segments, and the nine extension characters. It proves E.164, the sink and its bill, the outage against the rejection, and the refusal in production. I ran it once, and 53 of 53 assertions passed. `wisecoding_rich_narrated` is chapter 18's: the frame, the numbered options, the refusal of a number nobody offered, the checkpoints with a lifetime. `payments_webhooks_narrated` proves that an inbound signature is verified before it is believed: unsigned, forged, replayed. `sound_semantics_narrated` proves that the five values are the law's, that four sound and `muted` is silent, and that an invented value is refused. `sound_listener_narrated` proves the closed grammar, that no match is a result, and that the confidence is a separate call; I ran it once, and 21 passed. `sound_convergence_narrated` proves that the loop closes for the four states, that the language is negotiated per direction, that a sixth value is refused at declaration, and that a stranger's phrase gives no meaning.

A caveat for the court line. The sound guards need the engine's sound library, and for the loop, a voice and a recogniser. On a machine without them the voice guards print "skipped" and pass. A pass there proves nothing about speech. The guards for SMS and for the conversation have no such dependency.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

My study of the record counted the constituent units of messaging, eighteen in all, and what is missing. Its sentence is that the word for a messaging app appears in no code file and no design document of the library, and USSD in none. I re-measured it at this commit and confirmed it. There is no hit in the code or the design of base, core or max. The matches in the rest of the snapshot are word-list and test data files, and USSD has none in text.

The frameworks lead in understanding and in connectors. Chapter 18 states Softanza's side from the code: the conversation asks one slot at a time, over a graph the user must first have declared. It does not understand an open reply.

The field leads in channel coverage and in delivery truth, receipts and provider-approved templates. The SMS port says it itself: a sink proves what you would have sent, not that a carrier would accept it, and the live adapter is infrastructure-gated. No live client for SMS ships.

The voice plan: a declaration is portable, and a voice is not. A browser's voice has no buffer, so the loop closes on the native tier only. This machine can speak English and cannot hear it.

What the field has in cloud dictation and in trained recognisers for many languages, the record chose not to ship. Free dictation reached 66.7 per cent on the plane's own clean synthetic audio.

What the record claims instead: the dialogue is a rendering of a declared world judged by a court, deterministic by default. The medium is a port with a sandbox, so the conversation and its bill are rehearsed before an account exists.

The field leads in places, and the limit comes next, printed whole.

7. THE LIMIT

BUILT	the governed conversation with its question as data; the SMS port and its sandbox, with the bill per segment in two alphabets; a verifier for the body of an inbound message; five semantic values rendered as colour, earcon and phrase; a voice that speaks, and hears a closed grammar; a closed set of sentences a business agrees to say; a message of three fields between identities of one platform
DESIGNED	the commons service of messaging; conversational surfaces for a messaging app, SMS, USSD and voice, with a satisfiability matrix as a court; templates as a grammar
ABSENT	a messaging app, USSD or a telephone line, in any form; a conversational projection of a world; a channel, a session, a template, consent; the

> question in a language other than English; a voice for Hausa, Zarma or Fulfulde; the twenty-two verbs as a list the library holds; a reply with a space in it

My study of the record has two errors that this chapter's runs correct. The first: it says speech to text is missing. At this commit it exists, by decision, for commands and not for dictation. The second: it says no message exists. There is a message of three fields, between identities of one platform, and I show it:

HARO

```
oDb = new stzDatabase(":memory:")
StzPlatformQ("neighbourhood") {
        OpenCommonsOn(oDb)
        PostMessage("customer", "kitchen", "one brik, no egg")
        ? @@(Inbox("kitchen"))
        #--> [ [ "customer", "one brik, no egg" ] ]
}
```

✓ ran: printed what this block promises

It is a sender, a recipient, a body and a time in a SQL table. It has no medium, no status and no parts. It is the message my study found absent, in its smallest form.

What is built: the governed conversation; the SMS port and its sandbox; a body-signature verifier with replay protection, wired to no messaging provider; the semantic layer of five values rendered as colour, earcon and phrase. Also built: the voice; a minimal message in the commons; a closed set of sentences with a refusal; and the world's declared intents and reaches.

What is designed: `Provides(:Messaging)` as a commons service of the constellation. One design file says so, in two lines: a diagram that lists messaging among the commons, and a sketch that writes `Provides(:Messaging)`. The word `Provides` is in no code file of the application folder. The commons runtime as the operational counterpart of the declared provides and shares. Conversational surfaces for a messaging app, SMS, USSD, voice and web chat, the satisfiability matrix as a court, templates as a grammar, a channel in the platform profile: these

are proposals of my study of the record, written by a session, and not the library's design. A spoken analysis and a spoken transport: demos and compositions, and not a unit.

What is absent:

- Ports for a messaging app, Telegram, USSD, and the telephone call: zero hits. A USSD session has no shape in the library.
- A conversational projection of a world. The only projection is `Generate`, which knows three surfaces and refuses a fourth. No shell renders a question or a flow.
- The messaging units: channel, session with a window or a timeout, template, consent. No unit of consent exists for messaging. The word *consent* appears in the snapshot for the microphone's boundary, in comments, and in two design documents in other senses.
- The twenty-two verbs as a library list. `stzApp` holds five of them as five screen-intent methods, and the helper behind them does not check the word it is given. The other seventeen are in the constitution and not in the library. The bridge sentence is applied by hand today.
- The question in a language other than English. `AskIn` is English, and the tutor shows the pattern in four languages. In my research run at this commit, `NaturallyIn` asked for Zarma raised the opaque "Unsupported object type!" and not "no pack for this language"; no block of this book records that call.
- A voice for the languages that the product designs name, Zarma, Hausa and Fulfulde. None here; the library would refuse.
- **A reply with a space in it, from a phone.** In my research run at this commit I ran `ReplyIn("c", "grand couscous")` and it raised, with an uninitialised variable. With a hyphen it was admitted. No block of this book records either result, as chapter 18 says of the same fault. This is chapter 18's known limit, and it is decisive here, because people type spaces in a text message. The conversation above works only because the replies are digits and one-word dishes.

- Live adapters for SMS, for a messaging app and for a telephone line; delivery receipts; sender-ID rules.
- Nothing in the record runs on a real phone as a conversation.

`On(:whatsapp)`, `On(:ussd)`, `On(:voice)` and the satisfiability matrix are what my study of the record proposed, and I do not present them as what I mean to build. I found no decision of mine to build them in the record, and the limit is the list above, in which none of them exists.

That is the limit, printed whole; the six doors follow.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	The question comes back as data. You write the rendering and the reader for your medium in a dozen lines, and every reply goes through the same door whatever carried it.
The analyst	The goal is the schema of the conversation. The transcript says which message produced which fact. The sandbox counts the bill before a provider exists.
The designer	Your user keeps the phone they have: a text line with numbers, or a spoken phrase. The constitution's verbs are the grammar you render, and a verb the medium cannot render should be a red cell, and not a silent omission. That is designed and not built, since the matrix does not exist.
The decision-maker	A conversation by SMS can be priced in a test: two Arabic dish names moved a 93-character question from one segment to two. The question to ask a vendor, what refuses a sentence your business did not approve, has an answer here: a closed set.
The educator	The tutor already says the missing step as a question in four languages, in templates that carry no code. The record does not name a learner on a plain phone, and nothing serves one yet: there is no USSD and no messaging app.
The agent	You may propose a reply, and only a sentence of the closed set passes. An invented one is refused with its reason. And you cannot give a meaning to a phrase: the crafter declares it.

9. EXERCISE

Take a question your own world asks. The one-relation world of chapter 18's exercise will do. Before you run anything, write under each line the output you expect:

1. the number of segments of your rendering of the question, with names in the Latin alphabet;
2. its segments if one option is a word in Arabic script;
3. what happens when the customer's reply is a digit that was not offered;
4. what happens when the reply is "yes please".

HARO

```
? StzSmsSegments(YourRendering)[:segments]
#--> your prediction
? StzSmsSegments(YourRenderingWithOneArabicWord)[:segments]
#--> your prediction
```

✎ not run: an exercise, written for you to run

Run it, with the sandbox and never a real send, so that the exercise touches nothing. If a promise breaks, do not change it: explain the output first.

I left this one in your hands. The fourth prediction is a defect I did not fix, and section 7 lists it in bold: a reply with a space in it, chapter 18's known limit. Take the block of section 3 that sends a reply to a question with no options on the table, change the reply "2" to a text with a space in it, and run it:

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])
```

✎ not run: an exercise, written for you to run

Look at what the call does, and do not repair it. Section 7 says why it matters on a phone. When you have explained the output, you will know why the conversation above works only because its replies are digits and one-word dishes.

The next chapter is the environment. It declares the interface once, and every shell is a projection of it; a projection, like a medium, may add renderings, never verbs.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stznatural-narration.md`
NARRATION	`stzApp-building-your-first-app-narration.md`
ON THE SITE	zui
RULINGS OF THE STUDY	12.15, 12.23, 12.24, 12.25, 12.26, 12.27

PART V · CHAPTER 26

The environment

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 10 of 10 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take the shape the field's environments have settled on, a toolbar, a sidebar, an editor and a status bar, and I set against it a pattern in five marks. I run the pieces the library has under those marks: a shell declared as text and judged, two projections, a ribbon in the order of a life, a rail that proposes, a status bar that tells the truth. I say what I decided, to carry both shells, and what I refused, among it a flag that turns the accessibility tree off. And I say where it stands: at this commit, an IDE of this family is absent, and the pieces beneath it run. The principle is one sentence: declare the interface once, and every shell is a projection of it that may add renderings, never verbs.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field's environments have settled on a shape. A toolbar and a menu at the top, a sidebar of files, an editor, a terminal below, and a status bar at the foot. Lately an assistant sits in a pane or a terminal beside the code, and the human approves its edits. I take that shape from general knowledge, and not from the record.

The record's own field studies, dated, are three. The notepad that ships with the former host language, measured on 12 August 2026 at thirty-three source files and about four thousand lines, seven dock widgets, a fifteen-key configuration and no language service. It exists, it ships in the box, and it works on the day you install it. The fast editors, in the ergonomics specification of March 2026, which sets for its own product: input to screen under sixteen milliseconds on any modern hardware,

no exceptions. It wants minimal chrome, no status bar clutter, no toolbar forests. It asks for a dry run before a structural change, and for inspections that say why and offer a fix. And the status bar, as that specification wrote it: twenty-four pixels high, carrying only essential state, with five build states, one of which is *unchecked*, for a project where no build has run yet. The state of "not yet judged" has its own word. A bar that was green by default would be the field's usual lie.

Where the field is silent, the chapter answers. The field orders its toolbars by the tool's features, and not by the life of the thing being made. And it lets an assistant's proposal land as an edit, with the human's approval as a habit; that too is general knowledge.

2. THE FIRST PRINCIPLE

Declare the interface once. Every shell is a projection of it, and a projection may add renderings, never verbs.

The library's header for its authored interface says it: the file is the contract, and the native markup and the web page are its projections. My study of the record reads it as one declaration, two projections, proved to agree, resting on my ruling of 7 October 2026: carry both. The companion sentence is the interface constitution's, on the site: a new medium adds renderings, never verbs.

3. THE SOFTANZA FORM

First the stage, before the blocks. At this commit, **an IDE of this family is absent**. No class in the library is yet named for a ribbon, a rail, a spine or a status bar. My study of the record put it in a stage line proposed for the site: no IDE of this family ships today; one shell does, for the sound plane. The foundation that would make the next one a projection is specified and not yet built. I add one thing: the library generates one browser surface, the emulator's, and I show it below with a loss.

What the library has are the pieces under each of the pattern's five marks. The five are one shell, two projections, a facet ribbon ordered by the life of the solution, a rail that proposes and a status bar that tells the truth. They run, and I take them in

order.

ONE SHELL, DECLARED AS TEXT, JUDGED BY A COURT

The chrome is declared as text and judged before it is shown. The ribbon is generated from data, here a plain list; the facet as a library object is designed and not built. The four facet names are four of the five phases of the design corpus in the record: define, architect, build, govern, ship. I took four to keep the block short. Every declaration must carry its rationale, and a declaration without one is refused. Which element serves no verb? Run it:

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" ]
}
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

`Orphans()` answers `[ "work" ]`, and `IsClean()` answers `1` over the `12` declarations.

The generating loop is the book's own, and the roles are from the closed list in the library: `tablist`, `tab`, `group` and `status` are on it.

THE TWENTY-TWO VERBS

Every element must trace to at least one verb, and a word that is not one of the twenty-two is found. There is a division of labour. The library records the claim, with `SATISFIES`, and finds the untraced elements, with `Orphans()`. It does not hold the list and does not refuse an unknown intent: the guard asserts that an intent nobody has defined is not an error. So the list below is typed from the constitution, which is in its own repository, and the book holds it. Does the court refuse an intent that is not on the list? Run it:

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" ]
```

✓ ran: printed what this block promises

With `SATISFIES focus` on `work`, `Orphans()` answers `[ ]`, and so does the loop over the book's list. Asked about `transfer_funds`, `IsClean()` answers `1`! The loop over the book's list answers `[ "transfer_funds" ]`.

`work` was a pure layout box, and the court counts containers too. Whether a container should be exempt from the verbs is the constitution's to say, and not the library's. The families, from the constitution: orienting, attention, information, selection, action, and continuity. And the interpretation I give: the library finds the stranger, and a list the library does not hold cannot refuse it. Chapter 25 said that the bridge sentence is applied by hand today. I tell the removal as the refusal at the centre of this chapter, in section 4: the library does not own the verbs, the constitution does. The library's own design text still says twenty-four; it is not the list. The twenty-two are printed in the block above, typed from the constitution as Appendix C prints them.

TWO PROJECTIONS

The same text yields the native markup and the web page. Every declared element is in both, and the two spell a property differently exactly where the profile records a divergence. How many names are in both? Count them:

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
```

✓ ran: printed what this block promises

Eleven elements and not twelve: the panel is the page itself, and not a `div`. The last two lines show where the two differ. The native markup holds `--rmlui-direction` (`1`), and the web page does not (`0`). The native tier lays out through the engine's layout library, vendored, and draws through the graphics plane. I did not open a window.

THE BROWSER SHELL

The browser shell is a page that the language serves, on the loopback. One request, one process, no network beyond 127.0.0.1. A second reactor plays the client, so it is offline by construction. The handler reads a global, because lambdas of the host language do not capture locals. Where does the page live? Ask the server:

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()
```

✓ ran: printed what this block promises

`Host()` answers `127.0.0.1`. The cost of this shape is named in the record. A locally bound port is a surface and must be treated as one. In the charter's words, the design intent has been measured for a sound studio and not for an IDE. The library's backend already refuses to leave the loopback unauthenticated, at bind time; the bare server above does not check. What does the backend do with `0.0.0.0`? Try it:

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
```

✓ ran: printed what this block promises

It raises! `bRaised` answers `1`, and `IsLive()` answers `0`.

A STATUS BAR THAT TELLS THE TRUTH

A status cell is a function of the court's verdict, so it cannot say *clean* about a document the court refused. The function below is the book's own, over the library's `IsClean()` and its diagnostics. The status-cell class is not yet in the library, and the guard that a cell shows only what the court emitted is designed. The specification also names the state before any build, *unchecked*, and my function does not carry it. What does the cell say for a child declared nowhere, and for a role that is not on the list? Run it:

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]
```

✓ ran: printed what this block promises

The cell answers `refused: DANGLING_REFERENCE` and `refused: UNKNOWN_ROLE`, and *clean* only for the document the court kept.

And what a screen reader reads is emitted from the declaration. A region declared `ROLE status` is emitted into the accessibility tree with its rationale as its description, in reading order, and no flag turns the tree off. The layout needs a font file, and this block uses the system's. What does the tree say of the status region? Ask it:

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"' ]
}
```

✓ ran: printed what this block promises

The tree answers `status`, and for its description `The truth of the build.`, the rationale word for word! The library's header says why: every toolkit gates accessibility behind a performance flag, which is where users silently get nothing. So here there is nothing to gate and nothing to forget to switch on. The same file says what a screen reader can operate, an AI agent can operate.

A RAIL THAT PROPOSES

A rail is a view over what the world proposes. A pulse proposes and never relates. An act on the world clears a proposal, and the world's own status line counts from the same list the rail shows. What does the world propose, and what remains after one act? Run it:

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
}
```

✓ ran: printed what this block promises

`Live()` prints its line, `2 proposal(s)` at its end, and the judge compares what is printed. Before the act, `Pulse()` answers `0` and the rail still holds 2 proposals. After the act, one line remains: `propose visit for binta`. "2 thing(s)" counts the two things, Client and Visit, and not the instances. Chapter 20 owns the law beneath the rail, the proposal, the plan and the court.

And a screen is an intent. One place where the world speaks in verbs is its screens. Five methods exist: `ToDiscover`, `ToUnderstand`, `ToFocus`, `ToSelect` and `ToAct`. A sixth verb, `ToCompare`, does not exist. What answers when a screen asks for it? Run it:

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
```

✓ ran: printed what this block promises

The runtime answers, `Error (R3) : Calling Function without definition: tocompare`, and no court does: the intent word is stored as a string, and is not checked against any list.

That is what runs; now I say what I decided with it, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I decided to carry both shells. On 7 October 2026 I was asked whether the foundation takes the native shell or the browser shell, and I answered: carry both. The consequence, as my study of the record states it, is that one chrome is declared once, and two shells project it.

I decided that the mechanics are the library's and the IDE is the product's. The library's design document: the paradigm mechanics are platform; the book, the brand, the IDE are product, and never leak. The foundation ships refinement as a library verb available to every program.

I removed the interaction engine. On 26 May 2026 my commit removed twenty-two modules invented by an AI agent that were not in the original Softanza, among them the interaction module. I read the date, the count and the list in the library's git history, commit `b70f3ad91`, and not in the pinned tree. Its design text, which says twenty-four verbs and laws enforced by the engine, is still in the tree. The constitution's list is twenty-two.

I refused to resolve intents inside the language of the panel. The panel declares no meanings: there is no `:Danger` and there never may be. The intent slot was left deliberately unresolved. Above, `IsClean()` answered `1` for `transfer_funds`: the panel judges no meaning.

I refused to author the markup, and I refused a script engine. The native markup is emitted, never authored; there is no script engine; flexbox only. That is the reason the chrome is a text file and not a class.

I refused a flag that turns the accessibility tree off. The tree block above printed `status` and `The truth of the build.` without a flag to set.

I admitted a lens that cannot capture. The estate's charter for its products: one truth, in files, and two lenses, which in the charter are its two products, the front-door IDE and the commercial studio, and not two shells. What the charter refused was frameworks that capture the work. A lens that cannot capture is admissible. The environment is a projection of plain text, and not a store.

A session removed a facet, and I restored it. The reference design records that the Connections facet was first-class in the prototype of June 2026, and was removed by an AI agent on circular reasoning. The removal cascaded into five documents. I reinstated it. The ribbon is mine to decide.

I accepted the browser shell with its cost, unmeasured. The charter of the front-door IDE chose a browser tab, built with the engine, with no install step, and named the loss.

The pattern was learned across six efforts in two years, one of them built, by my study's count of the record.

Next comes the court, with the guards behind these pieces and the ones that are missing.

5. THE COURT

`gui_stzui_narrated` proves the closure of the panel court, a dangling reference, a duplicate, an orphan box, a missing panel, the mandatory rationale and a round trip. In scene 13 it proves the intent slot: `SATISFIES`, `Orphans()`, and that an intent nobody has defined is not an error. `gui_profile_narrated` proves that the profile is data, with three recorded divergences. It proves that one document has two projections and every declared element is in both, and that the fixture emits a self-checking page. The browser half of that guard is a procedure to reproduce. `gui_tier_agreement_narrated` proves that one display list rendered as vector and as pixels agrees to the pixel. It is not HTML against native.

`gui_accessibility_narrated` proves the tree emitted from the declaration. `05_stzapp_live_narrated` proves that a live world proposes and acting clears. `emulator_narrated` proves the generated surface of the emulator.

Three guards are to come: one that judges a shell against the twenty-two verbs, one that a status cell shows only what a court emitted, and one that says the two projections of a chrome agree in the browser. My study of the record lists them as absent, and the tree at this commit agrees.

The two-projections check can fail, and it did. The library's fixture is a page and not an assertion: it carries the web projection and the boxes the native layout produced. A script compares them and prints a verdict strip. A runner cannot judge it, so I read it by hand on 8 October 2026. On the chrome of the first block, eleven boxes with text in each region, the strip read "AGREE on every declared box of 11". Its note said that four boxes of automatic width differ, which is two shapers and not the profile. On the same three-band chrome with empty regions, the strip read "DISAGREE on 1 declared boxes of 3". The cause is in the web projection: an empty box is emitted as `<div id="body"/>`. An HTML parser reads that as an open tag, so the siblings that follow nest inside it. The native tier is unaffected. The fixture catches it! The library's guards do not yet, since their own showcase has no empty box: adding one is the next step. An empty workspace is the pattern's first state. So the strip tells the truth both ways, which is the chapter's fifth mark in one picture, and the web projection has a case to correct on exactly that state.

From the court I turn back to the field, and to where it leads.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The fast editors set a bar that the record writes down as law for its own product: input to screen under sixteen milliseconds, a cold start under four hundred, no exceptions. The record has not yet measured it on either shell: a latency number is absent, and it is the next step.

A text change dirties the whole document. One bound value changed re-shapes every string, where a colour re-shapes one. The field's editors patch, and lead here today; that is general knowledge.

The browser costs what it costs: file access through the browser's model, a port to guard, and an unmeasured latency. The native shell costs a window system, and a GPU for the raster path.

The field has a language service, as general knowledge has it, and the record has a contract; the service is the next step. The world graph, `stzWorldGraph`, is the explorer's model: it reads the world file the service would emit, and it is loaded by no entry file: a block that calls it without a hand-written load fails.

The field's fast editor treats the status bar as clutter. The pattern keeps one and asks it to be true. That is a bet, and I hold it as a view, not a result; the record's own corpus argues the field's side in the same breath.

That leaves the limit.

7. THE LIMIT

BUILT	the interface as a text with a court that finds the element serving no verb; two projections of it, native and web, with a recorded table of divergences and a page that can say DISAGREE; the accessibility tree emitted from the declaration; a page the language serves on the loopback; a world that proposes and whose proposals clear when it is acted on; screens declared as intents; one generated browser surface
DESIGNED	the shell kit with two shells; the chrome as closed kinds (spine, ribbon, rail, status bar), none yet a class of the library; the facet as data; a rail bound to focus; status cells read from the courts, with a guard that a cell never shows a state the court did not emit
ABSENT	an IDE of this family; a list of the twenty-two verbs in the library and a refusal of an unknown verb; a guard that judges a shell against the verbs; an editor; a measured latency

The web projection does not yet carry roles, labels and intents. `ToHtml` emits an identifier, a style and the text only. The tree that carries roles is a separate artefact, as the block above shows, so a browser shell would infer from

bare `div`s what the native shell emits. The plan itself says that the accessibility library it adopts has no web adapter.

The library's own mission control prints constants where the pattern asks for verdicts. The emulator generates a browser surface, a parts grid with status chips and fidelity badges, a device window and a console for each part. Its status words are constants of the device class and of the template, and they are not measured. Here a plan with one firmware part and one GPIO capability, and no pump anywhere. What does its page say? Build it and read it:

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")
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The part gets "2 approximated" and, in the grid's detail, a text about pump timing and sensor noise: the greenhouse demo's text, returned for any microcontroller. The header says "parts healthy" and the banner says "Ready to ship." for the same plan, from literals! A backend-only plan prints "parts healthy" and "Ready to ship." too, and neither "needs a look" nor "2 approximated" (read in the source,

not run). "Deploy to production" is an alert. The truthful half is real: fidelity is printed, and the console refuses a verb that the part's engine subset lacks. That is narration in the library, which I did not run.

This is the loss against the library's own page. It stands beside "a status bar that tells the truth" of section 3: the pattern asks a status cell to be a function of what a court emitted. And the library's own page prints its words from literals. The request to derive those words from the plan is open, and it is the next step.

`ToPanel` needs a font file and raises without one. The accessibility tree and the fixture need a laid-out panel, so the blocks that use them depend on a system font.

`Orphans()` counts containers. Whether a layout-only box must serve a verb is the constitution's rule to state.

What is built: the interface as text with a court; four closed kinds, a closed list of nineteen roles, a closed field set, a mandatory rationale, a round trip to a fixpoint. Two projections, with a recorded table of three divergences and a page that can say DISAGREE. The accessibility tree, and a bridge that reaches a real screen reader's interface on Windows, read and not run by me. A declared value changes and the screen follows, with focus kept. A page hosted on the loopback, and a backend that refuses to leave the loopback unsigned. A world that proposes, screens declared as intents, one generated browser surface. And one binary shell, the sound plane's listening bench, in about seven hundred lines of Zig, which I read and did not build.

What is designed: the shell kit with two shells; the chrome as closed kinds named for the spine, ribbon, rail and status bar, none yet a class of the library. The facet as data, with its group, its place in the life, its artefact kinds, its court and its profile. The rail bound to focus, and the governance cells read from the courts, with a guard that a cell never shows a state the court did not emit.

What is absent: an IDE of this family. A list of the twenty-two verbs inside the library, and any refusal of an unknown verb: `transfer_funds` passes the court. A guard judging a shell against the verbs. An editor. A measured latency. I tell the

pattern as designed, with the running pieces as evidence: the court of the chrome, the two projections, the accessibility tree, the page hosted on the loopback. No IDE of this family ships today.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You write the shell as text, and a court tells you the one box that serves no verb before anything is drawn. The same file is a native window and a page, and the library has a page that says when they disagree.
The analyst	The ribbon is the order of your solution's life, and not of a tool's features. The status cell reads the court, and you can ask the world what it proposes before it is asked to act.
The designer	Your interface has a constitution of twenty-two verbs and a closed grammar that adds renderings and never verbs. Every element must say which verb it serves, and the library already finds the ones that say none.
The decision-maker	Ask of any environment three things. Does its status bar read a court? Does its assistant live in one panel and propose? And who judges that the two shells agree? Here the answers are: designed; designed, with the rail's law built beneath it in chapter 20; and a self-checking page, which caught the empty-box case when I read it by hand on 8 October 2026; no guard judges it yet.
The educator	The exercise is a prediction: say what the court will answer before you run it. The status word that is a constant is a lesson in the difference between a label and a truth.
The agent	The shell is a text file that you can read and write, and the tree a screen reader reads is the tree you may read. Your proposals appear in a rail and commit nowhere.

9. EXERCISE

One exercise, in three moves.

1. Write a shell of your own in five elements: a title bar, a ribbon of three facets taken from your own solution's life, a work area, a rail, a status bar. Before you

run anything, write on paper which of the twenty-two verbs each element serves, and which one of your boxes you expect the court to call an orphan.

2. Run `IsClean()` and `Orphans()`, and compare with your paper. Then break the file three ways: a child that is declared nowhere, a role that is not on the list, and an intent that is not one of the twenty-two. For each, write what your status function will answer, and whether the library, the book's function, or nobody will catch it.
3. Emit the page, open it, and write the line the fixture strip will print for a version with an empty work area. Then look. The page comes from `ToFixture`, which takes the box that `BoxOf` reports for each name on the laid-out panel from `ToPanel`; `gui_profile_narrated` has the loop, and a font file is needed.

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() )
```

✎ not run: an exercise, written for you to run

If a promise breaks, do not change it: explain the output first.

I left this one in your hands. The `ShellStatus` of section 3 is the book's own function, and I did not guard it against the case below: the one section 3 ran through `IsClean()`, and section 7 lists as absent. Run the line and read the word the cell gives. Then make the cell refuse a stranger to the twenty-two, as the book's loop does, with the loop over the list that section 3 already holds, and without asking the library for a list it does not own (section 4 says why).

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"))
```

✎ not run: an exercise, written for you to run

When the cell is right, you will know what the library finds, and what only a list it does not hold can refuse.

Every shell of this chapter is a projection, and chapter 27 goes beneath them all: three layers, one engine, Haro. There every feature is implemented once, the surface is the promise, and the runtime is the part you measure, and never assert.

WHERE THE EVIDENCE LIVES

NARRATION	`stzApp-building-your-first-app-narration.md`
NARRATION	`stz-emulating-the-whole-solution-narration.md`
NARRATION	`stz-system-dev-to-deploy-narration.md`
ON THE SITE	zui
RULINGS OF THE STUDY	12.12, 12.23, 12.24, 12.25

PART V · CHAPTER 27

The engine beneath: three layers, one engine, Haro

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 9 of 9 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I load the three layers of the library, one entry file at a time, and count what each holds. I read the engine beneath them, its eighty-nine libraries and the functions it registers, and I let the court say whether it is optimised. I set down the decisions, that the future is Haro and that every feature is implemented once, in the engine, and I print the rows where a fast peer still leads. The principle stands over all of it: the surface is the promise, and the runtime is the part you measure, and never assert.

1. THE CONCEPT, AS THE FIELD HOLDS IT

A platform inside one codebase is drawn as a stack with one rule: an inner layer knows nothing of an outer one. The cleanest peer, as my study of the record puts it, is Rust's three crates for three depths of host, chosen by the target; Rust enforces the rule by its compiler, which is general knowledge of the field. Softanza holds it by entry files, and section 7 says what that costs.

The field's habit for an engine is a native library with a plain C interface, wrapped thinly by each language. SQLite is the common example, well established in the field: one C library, bindings in nearly every language. The record states the same design for its engine: language clients are thin bridges that add syntax and paradigms, and never computation.

And the field has done, repeatedly, a third thing: a language keeps its surface and its libraries and changes the machine beneath. The record's own example is a typed descendant of Lua. Another, well established in the field, is the alternative implementation of Python. Both teach the same thing the record teaches: the surface is the promise, and the runtime is the part you must measure.

The principle that follows is one sentence, and section 2 sets it down.

2. THE FIRST PRINCIPLE

Every feature is implemented once, in the engine. The surface is the promise, and the runtime is the part you measure, and never assert.

The first sentence is the engine architecture's design rule, in a session's document: every feature is implemented once, in the engine. Language clients are thin bridges that add syntax and paradigms, never computation. Chapter 4 told my ruling that follows from it: I refused computation in the faces. The second sentence is the brief of the runtime beneath: performance and portability, measured, never asserted.

And the definition that governs the engine's dependencies, which chapter 4 already states: sovereignty is nothing anyone else can withdraw. In the vision's words, it never meant every line ours.

3. THE SOFTANZA FORM

Three claims, and a block for each of what a block can prove. I tell you first what a block cannot prove, because it matters. The dependency rule of the layers, that no core file loads base and no base file loads max, needs a scan of about eight hundred and seventy load lines. Only a guard could carry that, and none exists. I did the scan with a script of my own, and section 7 gives its result.

A LAYER IS THE FILE YOU LOAD

How many classes does each entry file load? Three entry files, three fresh processes. Each prints how many classes it holds, and whether `stkList`, `stzList` and `stzListXT` exist. The block spawns child processes, because this chapter's

own process has already loaded the largest layer, and only a child can show the core alone:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The first process prints `17 1 0 0`. Core holds 17 classes, ten with the prefix `stk` and seven with `stzCore`, and no base class. The second prints `674 1 1 0`: base holds 674 and no max class. The third prints `714 1 1 1`: max holds 714. Each layer includes the one beneath and none above: core inside base inside max, loaded from below. A load of the core entry alone takes about a tenth of a second on this machine. A load of base takes between three and eight seconds, in single runs on 9 October 2026. My notes hold those times, and no block shows them.

The contract says base builds on core, by inheritance or delegation. What do the classes say? `ringvm_classesList()` returns, for each class, its name, its identifier, and its 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]
```

✓ ran: printed what this block promises

The second line prints `stzobject`: `stzString` inherits `stzObject`, and not `stkString`! Core carries a parallel hierarchy, in which `stkString` inherits `stzCoreString`, the first line's `stzcorestring`. Max does build on base: `stzListXT` inherits `stzList`, the last line's `stzlist`. Exactly two base classes inherit a core class, `stzSciNumber` and `stzPointer`, and one delegates, in a line of `stzPointer`. They are two among the hundreds of base. The contract is not met, and it is not met in the way chapter 4 asked about. I print it as a debt.

And the third layer by its prefix. In the process of the largest layer, how many classes begin with `stk`, how many with `stz`, and how many with `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 ]
```

✓ ran: printed what this block promises

The block prints `[ 10, 693, 0 ]`. Ten, six hundred and ninety-three, and none! The three do not add up to the 714 of the first block: the classes that remain carry none of the three prefixes, and a count by prefix says nothing of them, which is the

exercise's point. The forty classes max adds to base all carry the base prefix. The thirty-two classes with the third prefix exist, all in one folder, and no entry file loads them. Section 7 says it again.

ONE ENGINE

The layers are real at the binary. The build file declares the core domains and the base domains and builds one library per domain, in a loop. How many libraries does it declare for each layer? This block reads that file, and the claim is about that file:

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
```

✓ ran: printed what this block promises

this block touches files; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The block prints `4`, `85` and `0`. Four core libraries, eighty-five base libraries, and no library for the third layer: eighty-nine in all. The marker `fn addSpeech` is the first function after the two tables, a boundary in plain text, arbitrary but visible. A court of layers would parse properly, and I list it among the requests. My study of the record says eighty-six base domains at this commit, and I read eighty-five.

The engine tells the library what it is. Its version is a constant. How many distinct functions does it register in the virtual machine once it is loaded?

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
```

✓ ran: printed what this block promises

The block prints `0.8.0` and `2786`. Two thousand seven hundred and eighty-six distinct engine functions, the library's own view, taken at run time. The same call returns more entries, since the virtual machine's own are included, and several names appear more than once; I did not trace why. I assigned the list to a variable first. The call is re-evaluated at every turn of a loop, and I stopped two runs that grew to gigabytes before I learned it. The count moves with the built engine.

There are three counts of three different things, and the chapter says which it prints. The first is the count to the library at run time: the 2,786 above. The second is the count to any host through the C interface. A script of mine read the export tables of the eighty-nine libraries on 9 October 2026 and found 1,710 named symbols, of which eighty-nine are the loader's hook, so 1,621 functions. The third is the count in the Zig sources, as registration names: 2,703 distinct literals. The site says 752 and 2,652, at commit 010743cce as the page reads on 9 October 2026, and the library's typed constants say 1,513, dated 16 June 2026. I cannot reproduce 752 by any method I tried. Print only a count the build reads, with its method and date.

The engine does the work. What does the host's `upper` make of the German sharp s, and what does the engine's case mapping make of it, its own table over a vendored library? A second vendored library stems words. These are two of the engine's nineteen vendored libraries, which section 4 counts:

HARO

```
? upper("Straße")
#--> STRAßE
? Q("Straße").Uppercased()
#--> STRASSE
? @@( Q("running runs ran").StemmedWords() )
#--> [ "run", "run", "ran" ]
```

✓ ran: printed what this block promises

The first line prints `STRAßE`: the host leaves the sharp s alone. The second prints `STRASSE`: the engine writes SS! The third prints `[ "run", "run", "ran" ]`: the stemmer folds `running` and `runs` to `run` and leaves `ran`. If Haro's `upper` becomes full Unicode, the promise changes.

And the engine is optimised. The default mode of the build is optimised and safe, since commit 1fdc2eda5, which the library's git log dates 26 July 2026. The pinned files hold the commit and not the date. Until then every library the library had ever loaded was compiled with no optimisation, and that closes the question chapter 22 left open about the build mode. A guard reads time, since it cannot read a flag. Do twenty sums over two million doubles that live in the engine finish in under a tenth of a second, and are they right?

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()
```

✓ ran: printed what this block promises

this block touches the clock; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The block prints `1` and `2000001000000`, the boolean and the sum, and no time, because a time is not a promise. In three runs on 8 and 9 October 2026 the elapsed times were 0.02, 0.04 and 0.03 seconds, from my notes and not in the run record. The guard's own table says 0.14 for the unoptimised mode and 0.04 for the optimised one. The guard itself ran on 8 October: seven assertions in the file, six reached here, six passed. The seventh sits on an else-branch that is not reached when the parser is built.

WHAT RUNS IT

What runs every block of this book? I ask the runtime for its version:

HARO

```
? version()
#--> 1.27
```

✓ ran: printed what this block promises

It prints `1.27`: the version of the former host language, the oracle against which the new runtime is judged. Haro does not run the library. Under a future runtime the promise changes, as the proof line says.

The decisions come next, and the first of them is about that future.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I decided that the future is Haro, and not the former language. My words of 7 October 2026, as my study of the record quotes them, the bracket mine: the future is Haro, not [the former language], and the site must be clear about it. Haro's face of the former language will be maintained only so that the code already written continues to work. What is not carried is that language's runtime and its governance. The first ruling, on 28 August, was that it is not carried forward at all; it was superseded two days after, as the vision's amendment says.

I decided that the rewrite is a rename. The code of my articles, the narrations, rewritten in the name of Haro, is the same as the former language's face. The charter measures the same: Haro's own stages run the same text, byte-identical on the new compiler and on the oracle.

I decided that Haro is the language of languages. The charter's header quotes my note of 1 October 2026. The charter was ratified that day, with my "go with your recommendations" on seven decisions. Its amendments of 4 October answer nineteen open questions of the record, and the version stays ratified until I have read its section 15.

I decided that the surface is kept. Its surface today is the one the compiler compiles. The library is its compatibility specification. Other surfaces are faces over one program graph: every syntax a face, so that an AI agent fluent in the former language, in Python or in JavaScript writes Haro. Of these only the first face exists. A Python-shaped face is a spelling, not Python.

I removed the old framework, with no fallback. Engine-only: no fallback, no feature flags, no Qt dependency. The status of 12 June 2026 says zero of them remain. I checked. No loaded file loads a Qt library. Two external extensions are loaded by name: a zip one, which base loads, and a cryptography one, in a folder that no entry file loads.

I directed that the language's own extensions be removed. My directive of 13 June 2026: ensure that no external extensions are required, and that engine alternatives replace them. The library still loads one of them, a zip extension, as the entry above says.

I put the engine in Zig, with no dependency on a library, as my study of the record reports my ask. It is qualified by the record's own definition of sovereignty: nineteen vendored libraries, each with a record.

I let everything go into base. My sentence, as my study of the record reports it: under the pressure of time, everything went into base, waiting for the right moment to refactor again. By my count below, base is ninety-three per cent of the lines.

I chose safe optimisation. Not the fastest mode, and not the unoptimised one. I chose it because a clean panic beats silent memory corruption.

I refused a headline that does not reproduce. The memo of 13 September 2026 says the claim to retire is that the runtime beats the fast peer, present tense, on x64. Chapter 22 told the refusal, and section 6 prints the rows.

I refused a percentage. The figure is mine, and it should not be printed without the guard that produces it. Chapters 31 and 33 say so.

I have no sentence on how a layer is chosen in Haro. My study of the record only proposes that, in Haro, layer selection should be a declaration the compiler honours. The charter has no text on core, base and max. Base inheriting core is the contract as designed, with the stage word designed. The code does not meet it today, as section 3 prints, and the book's account is what the code does: two implementations over one engine. And in Haro, a layer is chosen by the file you load, today. A declaration the compiler honours is my study's proposal, and is designed.

Section 5 reads what the court proves of all this, and what no guard proves.

5. THE COURT

`engine_is_optimised_narrated` proves that the engine is compiled with optimisation: twenty sums over two million doubles under a tenth of a second, and the right answer. It proves that an index of 99 and an index of 0 are refused and the buffer unharmed, and that the vendored parser which the optimised mode once broke now answers. I ran it: six of the seven assertions in the file were reached, and six passed. `engine_senses_narrated` proves that the engine gives the process five senses: memory, its peak, system memory, processor time, and a bounded series. I ran it: thirty-six of thirty-six, and the run then stops with "STOPPED!" by design. `wasm_narrated` proves that the engine's entry for the browser exports its curated interface and that the build has a target for it. It builds a subset and asserts that it is smaller. I did not run it, because it calls a build of the toolchain. `threat_model_narrated` reads the threat model and checks that each guarantee row names a proof and that each guard file it names exists. Rows G11, the fuzz harnesses, and G25, the bill of materials with its digests, concern the engine.

What does not exist: **no guard judges the layers.** No file under the test folders loads a layer's entry file or reads the contract. There are 529 narrated guards in base, and none in core or in max. My study of the record says 517 in one place and 387 in another, and the site 501; I print only what the build reads. And no guard judges the engine's counts: the numbers the engine's command-line status prints are typed constants dated 16 June 2026.

Section 6 sets the runtime beside the field, and the peer leads every row.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

Speed. The runtime beneath against a fast scripting runtime of the field, on 13 September 2026. The method is the memo's own: six pairs of kernels written fresh, identical algorithm and sizes on both sides, each side in its fast idiom, checksums across languages asserted before any number. The minima are over three inner repeats and two process runs, the new compiler built for speed, the peer interpreter-only with no JIT, on Windows, x64. The peer leads every row:

ROW	THE RUNTIME BENEATH (MIN MS)	THE PEER (MIN MS)	THE PEER LEADS BY
sieve	128	74.5	1.72
matmul	349	185.6	1.88
fib	57	50.2	1.14
mergesort	376	206.7	1.82
binsearch	536	513.1	1.04
bytescan	53	37.5	1.41

The memo's words: the peer leads every row, fib and binsearch are near-ties, and nothing is worse than 2x. ARM is unmeasured. The win that the memo keeps: within 2x on every row, which no version of the former runtime ever was.

The headline is still alive in the repository's own notes. As I read on 9 October 2026, the brief of the runtime's own repository and its notes still say that the spike beats the peers on every row. The correction is carried in another repository of the estate and in the memo. In the runtime's own record it is carried only for its release-candidate configuration, in one section of the notes. The memo asked for it. The memo of 13 September 2026 withdraws the claim, and the six rows above are its rows, although the repository's notes still carry the old claim. I name no peer, and describe it as chapter 22 does: a fast scripting runtime of the field.

Memory. On the same day the note says: resident, running the library, this morning 733 megabytes, now 149, and the former runtime 139. A loss of ten to the former runtime, after a fall of 584. A win and a loss in one note.

Compatibility, on 5 October 2026. The wide suite of the library was run on a snapshot. The note records parity of assertions on every one the former runtime passes, that most programs agree on every stable line, and that nineteen rows differ in kind. The note does not say Windows in that entry, and the Linux pass is listed as the next step. These are a session's notes from a private branch, and no number of mine. I print them, dated 5 October 2026, with the stage word in development: 8,540 of 8,540 assertions at parity, 1,253 of 1,253 tests that bear assertions, and 3,425 of 5,112 programs identical on every stable line. Another 1,267 are identical but for the line that prints the elapsed time, and 19 rows differ in kind: 13 print more, 5 are refused or raise, one writes a stderr line. They are the note's counts; the note does not place the remaining programs, and I ran none of them.

The platform. Built and run on Windows today. A Linux build of eighty-five of the eighty-nine modules exists on a working branch; the site says it is in construction. A whole program with its engine in one file, with no dependency, is absent.

The engine against the field. The threat model's row for fuzzing says it itself: no scheduled or coverage-guided fuzzing, no continuous integration and unsigned commits, which I deferred for the development phase. The field's maintained native libraries have continuous fuzzing and signed releases. And chapter 22's loss, unchanged: no collation by locale, and no time zones.

Section 7 says where the layers and the engine stop.

7. THE LIMIT

BUILT — three layers chosen by the entry file you load, each including the one beneath and none above; the dependency rule held at the loads; an engine of eighty-nine libraries, in Zig, built in optimised safe mode, with the engine's own count of what it exposes; Haro as a ratified charter and a spike whose courts run on the new runtime and on the old one

DESIGNED — the base layer inheriting the core layer; the max layer as a built layer; a court that checks the layers against their contract; a layer chosen by a declaration the compiler honours; the other faces of Haro over one program graph; a loader for one domain of base; the standalone engine repository and its command line; the engine's polyglot bridge, which is chapter 16's

ABSENT — a lean core layer that runs its strings; a Linux or macOS engine in service; a measured pass of the library on the new runtime under Linux; a `haro` executable; Haro running the library; a single executable with no dependency; an engine test count that I may print

The lean layer does not run. What does a string of the core give back when I ask it for its content?

HARO

```
o = new stkString("abc")
? o.Content()
#--> Error (R3) : Calling Function without definition: stkenginestringfrom
```

✓ ran: printed what this block promises

The runtime answers: `Error (R3) : Calling Function without definition: stkenginestringfrom`. The core string cannot give back `abc`! The core string class calls engine names that no engine library registers. In a process of the core alone, the engine's core library registers five hundred and thirty-four functions under the base names and none under the core names. That count is from my notes, and no block shows it. So `stkString` and `stkChar` find nothing. My study of the record files it as an open item of the core and the engine. `stkList` runs, which chapter 4 showed, and the load of seventeen classes in a tenth of a second is real.

The lean layer is not lean at the binary, for strings. By file size, on the build of 5 October 2026, the core string library is 22,355,968 bytes and the base one 22,362,624. The read-only data is 20.8 megabytes in both; I did not diagnose what it holds. The four core libraries total 24.1 megabytes of the eighty-nine libraries' 122.4. The core libraries for dates, files and locale differ from their base twins by under one per cent.

The dependency rule holds at the loads, with one exception at a call. I parsed every load of every non-test, non-archive file of core, base and max, 871 lines, and resolved each. Core loads, outside its own folder, only the engine face; base loads core and the engine; no file under base loads anything under max or future; max loads base. But `FindMin`, in the core's list of numbers, calls a function that is defined only in base. In a core-only process, `FindMin([4,2,9])` answers 2, and `FindMin("not a list")` raises "Calling Function without definition". In a base process it raises the intended error. I found one instance by a search for base names in core, and I did not look for others.

On a case-sensitive disk the core entry file does not load. Its second line loads a file by a name in the wrong case. My scan finds twenty such loads in non-test files at this commit; the runtime's own record counted fifty-five with tests, on 8 October 2026. The book is built on Windows, which hides it.

The third layer is written and not wired. Its thirty-two classes are all in one folder, and no entry file loads them. The first of them works when loaded by hand. I ran it and do not show it, a string of eleven characters counted by it. I tried none

of the other thirty-one. A library layer for them in the engine does not exist.

What is built: a layer is a folder, a prefix and an entry file. I counted on 9 October 2026 at this commit, by files with the extension, excluding test and archive folders. Core, 27 files and 6,366 lines; base, 984 files and 519,579; max, 115 and 31,292; and a folder of the future, 51 files and 6,356. Base is 93.2 per cent of the three layers' lines. The engine: eighty-nine libraries from the Zig sources, 401 source files, 179,985 lines, and 2,464 declared tests, which I did not run. The toolchain is Zig 0.15.2 by the build file's own comment. Nineteen vendored libraries, each with a record of component, version, source, licence and digest. The record's own gaps: an archive digest is recorded for nine of the nineteen, and three have no version.

What is designed: the base layer inheriting the core layer. The seven shared domains are implemented twice over the same engine. Max as a built layer. A loader for one domain of base. A court that checks the layers against their contract. The standalone engine repository and its command line, which the architecture lists as a phase. The engine's polyglot bridge, which is chapter 16's.

What is absent: a Linux and a macOS engine in service; a measured pass of the library on the new runtime under Linux; a single executable with no dependency; an engine test count that I may print.

HARO

I tell what is built, what is designed, and what is absent, as the record has it on 9 October 2026.

Built, as a charter, a family of grammars and a guard. The charter, version 0.2, ratified on 1 October 2026. A spike of nineteen steps, which the record calls rungs, between 1 and 4 October, each a grammar document with fixtures. Its guard reported a pass of the whole run on 5 October. Its courts are programs that run byte-identical on the new compiler and on the official former runtime. The new compiler also formats and dumps the library's files: of about six thousand, nearly all print the reference's bytes, and a few dozen are refused by both. A first slice of a graph of the code was built on 5 October.

What Haro is today, in one sentence the record supports: a ratified charter and a spike whose courts run on the new runtime and on the old one, over the surface the library is written in. There is no `haro` executable. The folder of binaries holds the compiler and two others. The forged courts load no library.
Haro does not run the library. The vision of 27 September said that Haro does not exist and that the new runtime's Linux row is unmeasured. The first half is out of date, since a charter and a spike exist. The second half is still true of the wide suite on Linux.

Designed: the three altitudes, declaring, declared, and computing; the six letters of its alphabet, which the charter names, five of them forged whole in the spike; refinement as the write surface; other faces over one program graph. Those faces arrive as printers before they arrive as parsers, and never as semantics. The verbs `haro forge`, `check`, `refine`, `run`, `project` and `narrate` are a recommendation, and not a program.

The runtime beneath, as the record dates it. Its brief: a register machine and a compiler from the former language's source to its bytecode, running existing programs. About three hundred thousand lines of Softanza are its compatibility specification. Its public face: the main branch is public and untouched; the work lives on a branch pushed to a private remote only. Its last commit at my reading is of 8 October 2026, at 22:26. The stage word is mine, and it is "in development".

The six doors follow, one for each reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You choose your depth by the file you load, and you can prove it in three fresh processes. The lean layer does not run its strings today: reach for base. Read the engine's count the way the library sees it, and not the way a page types it.
The analyst	Three numbers for "how big is the engine", and three methods. Ask which one a vendor means. The core library is the size of the base one for strings.
The designer	A layer is a decision about what you ship, and not about taste. At the binary, core and base have a library per domain and the third layer has none; the web subset, which chapters 4 and 24 show, is a subset and not a layer.
The decision-maker	The two questions of chapter 4, plus two. What runs it: version 1.27 today, with the replacement in development. What the replacement measures: within 2x of a fast peer on every row, by 1.04 to 1.88 times, which no version of the former runtime ever was, and ten megabytes of memory more than the old one. This is my view: a record that prints its loss is the only kind to trust.
The educator	Teach the dependency rule by making a learner load the core alone and predict, then count. Teach "measured, never asserted" with the memo of 13 September 2026.
The agent	You may count the engine's functions yourself. Do not repeat a count from a page, and do not write a loop over the list of classes directly. Tell the reader which layer a class lives in by its parent chain, and not by its prefix.

9. EXERCISE

Before you run anything, write three numbers: how many classes each entry file will load, and which of `stkList`, `stzList` and `stzListXT` each holds.

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().
```

✎ not run: an exercise, written for you to run

Then choose any class of the library, predict its parent, and read it with the parent chain. Write the line your prediction got wrong. A learner who guesses that `stzString` inherits `stkString` is in good company: the contract says so, and the library does not. I do not say which of the 17, 674 or 714 is which.

I left this one in your hands. The count by prefix in section 3, `[ 10, 693, 0 ]`, is correct, and it says nothing about layers. Write that count again so that it asks each class's parent chain, read with `ParentOf`, and not its first three letters, and look at what you get. Section 3 says what the prefix hides, and section 8, the agent's door, says why I refuse it. When your count answers, you will know why a class's layer is read from what it inherits, and not from what it is called.

The next chapter leaves the engine for a restaurant in Lyon, and for the one sentence the project there taught: deliver small, fast and useful, and let the foundations mature in parallel without ever blocking a delivery.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	architecture
RULINGS OF THE STUDY	12.10, 12.11, 12.15, 12.21, 12.28, 12.29

PART VI

Software crafters, forged

28 · A restaurant in Lyon
29 · A bank in Niamey
30 · An organisation in the field
31 · What the projects show, and what comes next

PART VI · CHAPTER 28

A restaurant in Lyon

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 5 of 5 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I tell a paid lesson in order, with its dates: a restaurateur in Lyon, two years of specification that produced nothing he could use, and the reversal of 4 July 2026 that put the customer first. I show what the library carries of each lesson: the Lock as a rule over a cycle's worlds, a payments sandbox that is not agreeable, a constellation that puts a world to sleep, a speed budget declared and judged. I say where it carries nothing, and the six things I refused, among them the grand system in one piece, silent drift, the agreeable fake, and crediting the owner with what a simulation said. The principle is one sentence on the site: deliver small, fast and useful, and let the foundations mature in parallel without ever blocking a delivery.

1. THE CONCEPT, AS THE FIELD HOLDS IT

Each lesson of this project has an ordinary home in the field, and the field keeps it outside the program.

A fixed scope for a cycle lives in a process. In Scrum the goal of a sprint does not change while it runs; in Shape Up, the time is fixed and the scope varies, and new ideas wait outside the cycle. Trying a service before paying for it lives in a vendor's test mode and in tools that stand in for an HTTP service. A performance budget lives in a continuous-integration check of a page's weight and speed. The web's own measure of speed is a figure taken over real page loads, at the seventy-fifth

percentile. The picture of a whole constellation of small applications lives in the super-app: one host that embeds a runtime and runs mini-applications inside it. All of this is general knowledge, not a study of the record.

All four are good practices, and in all four the program itself knows nothing of them.

2. THE FIRST PRINCIPLE

The project was a paid lesson, and the lesson is one sentence on the site: **deliver small, fast and useful, and let the foundations mature in parallel without ever blocking a delivery.**

Its companion is this chapter's rule of evidence, taken from the same page: a lesson is listed only when a file of the library carries it.

3. THE SOFTANZA FORM

A restaurateur in Lyon owns a couscous restaurant, and he wanted a platform for neighbourhood commerce. He is not a programmer, and the project's own notes say so. I am going to tell the project in order, with its dates, and then show what the library carries of each lesson.

It began with an offer, on 2 May 2024, for a mission of business analysis. What followed was two years, in the owner's own count, of research and reflection, and of what I called specification: founding analyses, a novel of narrations, a first product specified in full. It produced nothing he could use. In the middle of 2025 the project came close to ending, and a silence of six months followed. That is where I learned the doctrine of section 2: deliver small, fast and useful.

On 3 July 2026, two of us rehearsed the project: a scoping of his answers that I wrote with a session, and a campaign of twenty-two simulated tests, run by the sessions playing the owner. I say "simulated" every time, because the answers in that file are not his. The next day, 4 July, he answered a real scoping report in his own words. He reversed the order of the work. We had imagined a line that ran from the supplier to the restaurateur and ended with the customer. He turned it around: customer first, restaurateur next, supplier last. And he wrote that the

speed of ordering was his first priority, non-negotiable. His great fear was friction at the first load, and the second fear, in the same report, was the calendar: none of the deadlines until then had been kept.

The project's answer to his fear is a document and a notebook. We called it the Lock. In July, a second scoping proposed it to him, clause by clause. The first three: a single amendment; every new idea goes into a notebook, read at the end of the cycle; one entry needs one written exit. The last two: a demonstration every two weeks and no date beyond the next cycle; and a definition of done. He accepted four clauses as written and adjusted the fifth by adding a criterion of his own: that other merchants and partners must be able to install the product alone. In the record I am the one who asked for the Lock first, as a rule on myself: that the change of course had to be the only change of course allowed. On 4 August 2026, the first amendment closed. Twelve decisions were sealed; three were asked again in plain words, because he himself had flagged that he had not understood them. The lesson written down is about how to ask: the moment a question speaks of milestones and phasing and cycles, the owner stops following. So the questions moved into his trade's language.

On 15 August the first demonstration was designed for him alone. The checklist is titled with his name and "alone". He opens the delivery on his own computer, scans one code with his phone and another with his tablet, places his own order from the phone, and watches it land on the tablet. If he gets stuck, a button prepares a message to paste into a messaging app. The criterion of success reads: he orders from his phone, and it lands on his tablet, without us. The notebook shows it happened: his remarks of that day were entered in it on the 17th. And on 19 August he spent a field day with two phones and photographs sent over a messaging app, measuring on a real phone. A hotspot works with no card in it; the address of the gateway was stable across restarts. A phone is a Wi-Fi client or a hotspot, never both; an iPhone's system does not return by itself to a network with no internet. The project's notes then call him its co-architect. The record ends on 25 August 2026.

That is the story. What does the library carry of it?

The Lock. I have to be exact here. The Forged page of the site holds a rule I made: a lesson is listed only when a file of the library carries it. No file carries the Lock. It was a pact and a notebook, kept in documents. What the library can do is give you the nearest declaration of it: a rule over the worlds of a cycle, that a world without a written admission is a finding. What does the court say when an idea enters the cycle with no written admission? This block is a construction, not the restaurant's code:

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
}
```

✓ ran: printed what this block promises

The idea of a loyalty club arrives, and the cycle is no longer sound! The court answers `UNSOUND (errors present)`, and names the rule and the node: `ERROR the-lock @ loyalty-club`. The cycle is sound again when the idea carries its

written admission.

Try payments without a subscription. The project's rule was to test without committing. That meant a payment institution's free sandbox and the library's simulated services, and no subscription before a gate that the notes call the go-live gate. The library's side of it is a payments sandbox that is not agreeable. It behaves like a gateway: a decline is an answer, an outage is not, and a capture cannot be made twice. The amounts are integers in minor units, so 4200 is forty-two euros. What does it answer to a token it was told to decline, to an outage, and to a second capture? Run it:

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
}
```

✓ ran: printed what this block promises

Three lines printed, a decline, a refused and a reason, and none of them is an empty string: `declined`, `refused`, and `this authorization was already captured`.

And the production gate refuses to open while a fake is still bound, even for an entitled person. Here the live binding is an HTTP sandbox standing in for an adapter, as the guard does it: the registry holds only the name of the secret. What does the production gate answer when the owner, entitled, asks for production while the fake is still bound? Run it:

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
}
```

✓ ran: printed what this block promises

The owner is entitled, and he is refused! Nothing is committed, and the log says why: `REFUSED -- the external surface is not production-ready`; the reason under it is the rule: `a fake must never ship`. It opens when the binding is declared live: the gate reads the binding's kind, sandbox or live, and not the class behind it, which is why an HTTP sandbox bound as live opens it here.

The constellation. When he reversed the order of the work, the project's architecture was redrawn as a constellation of five worlds on a shared ground. The worlds are the customer's phone, the kitchen screen, the merchant's workshop, management, which is the first application kept and put to sleep, and the publisher's console. The shared ground is identity, the catalogue, orders, payments and the fleet. In the library, the last world in the order of work can be put to sleep, and it is revived when its turn comes. What does a call across a bond answer while the target world sleeps, and after it is revived? Run it:

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
}
```

✓ ran: printed what this block promises

The call across the bond passes. The call to the sleeping world is refused! The reason is one line: `target world 'supplier' is not active.` Revived, it passes.

The speed budget. The owner's first priority was speed, so speed had to be a promise judged by something, not an aspiration. I hold this as a view, not a result. The budget has three numbers: the time to first display, the time to react to a tap, and the time for an order to reach the kitchen. The library declares them next to the code as expectations and judges them against what was measured. A breach fails the same gate as a security finding. He tightened the thresholds himself, in his second scoping, and I print none of them. The name of the monitor, the bounds and the samples in this block are mine, to show the form; they are not the project's budget. What does a breach look like? Run it:

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)
```

✓ ran: printed what this block promises

The first check prints `3 expectation(s), last check: MET.` Then one order takes 3100 milliseconds, and the gate is no longer sound! The finding says it in the budget's own words: `p95-order.to.kitchen.ms-under-2500: measured 3100 (must be <= 2500)`.

Does the library carry what the project learned about interfaces? It does not. The project's file of interface rules is experience in a few lines, fifteen rules under one directive that the project calls the three-second test, that a restaurateur has no time to look at a crowded screen for several seconds: one clear action per screen; undo instead of confirm; a button that is refused is disabled in view, never hidden. And one test: a simulated first day on an empty database found the gravest hole, that no supplier could be created in production. Those rules were forged, in the project's own count, by eleven iterations and a campaign of twenty-two simulated

tests with seventeen fixes. The library carries none of them as a check. So by the rule of this chapter they are the project's lesson and not the library's, and I tell them here without a block.

Six refusals follow, each with its reason.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused the grand system in one piece. After two years of rich specification, the project nearly ended. The estimate of 2025 stays the reference for the project's scope; it is never to be presented as today's calendar. What replaced it is the doctrine of this chapter: deliver small, fast and useful.

I refused to throw the first application away. When the owner reversed the order, the application already built was kept, dormant, until the first version of the new order was in production. The project's own words for it: nothing is thrown away.

I refused silent drift. The Lock exists because a project that has failed once drifts without anyone deciding to. A second change of course before the first version runs in production is refused, and so is an idea that enters the cycle without a written exit.

I refused the agreeable fake. A double that silently answers with an empty string produces a test that passes for the wrong reason. The sandbox declines when a gateway declines, refuses when a gateway is down, and refuses a second capture. And I refused a fixed fee before the go-live gate: no subscription to a payment institution before the owner's written go.

I refused to credit the owner with what a simulation said. The project has carried a rule since July 2026, that no action on its screens takes more than two gestures. The site prints it as part of the first amendment, and the project's documents carry it as the owner's constraint. Its earliest source is a line in the file of simulated answers of 3 July, written by me and a session. The project set it for itself, from its own simulation of his answers, and kept it. I tell it that way. What he did say, and what I take as his, is the priority on speed.

I refused to say the library was born of his need. The constellation design, the virtualization plan, the registry, the payments port and the performance judgment were all first committed after the reversal of 4 July. The plan is dated 23 July, and the performance study 29 July. The site says carefully that the project needed these things and that the library carries them. I will say no more than that. The record does not say which came first, his need or the parts: it shows the project finding them in the library, and I leave the order open.

Five guards stand behind the five blocks above.

5. THE COURT

`06_stzsuperapp_narrated` proves the constellation: worlds joining, a call across worlds needing both worlds active, a declared bond and governance, and a retired world breaking its bonds and reviving. `service_delivery_narrated` proves that a production deployment by an entitled actor is refused while a fake is bound, with every reason logged, and that it opens once the binding is real. `payments_port_narrated` proves the sandbox's state machine: capture once, capture only an authorised charge, refund no more than captured, a decline is an answer and an outage is not. `judgment_narrated` proves an expectation declared in the operational vocabulary and judged against measured numbers, a breach as a finding that fails the gate, and an expectation on a number nobody measures as an error and never a silent pass. `graphrule_object_narrated` proves the rule as an object attached to a model, with the operators the Lock's construction uses.

The court proves what the library does; it does not say where the field does better.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in storage, in queries and in ecosystem. The library's own study of the performance field, written against the tools of early 2026, says it without softening: no time-series database, no query language for it, no dashboards, by design; export instead. Among the five properties it claims against them, the one this chapter uses is the first: nothing in the field couples a budget to the codebase's own rule system.

The field leads in the browser. The restaurant's worlds run in the browsers of customers and the kitchen, and I found nothing in the library that measures a first display or a tap there. The library's monitor observes a server, and the SLA judges

whatever numbers are recorded into a timer. The owner's measurement protocol is at the seventy-fifth percentile on a mid-range phone. The library's terms cannot declare it exactly: they offer the fiftieth, the ninety-fifth and the ninety-ninth.

The field leads on real services. A double is not the service, and the plan says so: you still need at least one real run before production, and recorded responses drift. The real providers lead in the edge cases the sandbox does not model.

And the super-app hosts lead in installed base. The design names what the industry's super-app has, a store, a review, a native bridge, and replaces it; it does not claim users.

Now I say where the chapter stops.

7. THE LIMIT

BUILT — the constellation of worlds with its bonds and norms; the payments sandbox and the production gate that refuses a fake; the speed budget declared and judged

DESIGNED — the Lock as a rule of the library; the constellation written in the near-natural idiom the design describes

ABSENT — the restaurant's live payment adapter; a measurement of the budget on a real mobile network; a speed expectation in the owner's own percentile; browser-side timing; the platform on the Softanza engine

What is built: the constellation of worlds with its bonds and norm-gated crossings, retirement, revival, hot swap and nesting; the payments sandbox and the production gate; the speed budget declared and judged; the graph rules the Lock's construction uses.

What is designed: the Lock as a rule of the library, of which the record has no code; the near-natural idiom of the constellation, with the verbs the design names and the class does not have.

What is absent: the restaurant's live payment adapter, which the library's charter calls a later task and the owner's decision. A measurement of the budget on a real mobile network, which the project's dashboard of 23 August postponed to the next cycle. The interface rules, the two-gesture rule among them, as checks, a seventy-fifth percentile expectation and browser-side timing, of which no document of the library holds a design. And the state of the platform after 25 August 2026, where the record ends. Nothing in it says the first version reached production at the restaurant, passed the go-live gate, or took a real order from a real customer. The record is silent on what happened after 25 August 2026, and I say so. Of the owner's tightened speed thresholds I print none, and say only that such a budget exists.

And what runs today is ordinary. The worlds are single HTML files, one per world, with a vendored framework, no build step and no content network. The server is a Node program of a few hundred lines. It uses five of Node's own modules, serves the worlds on the local network, and keeps one thing: a journal, chained by hashes, replayed at restart. The first management application has a phone wrapper around a web view. None of it runs on the Softanza engine. The library is the project's design reference. The constellation is, in the site's words, a design the library can run. The restaurant does not.

Six doors follow, one for each reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	You declare worlds and bonds and run a constellation in which a call to a world that is not yet active is refused with its reason. The restaurant's own code, however, is a web application.
The analyst	Rigour in numbers was the contract of the first application: a due date computed from the invoice date, a date required, a debt with no due date made visible.
The designer	Fifteen interface rules from one directive, the three-second test; and a refused button is disabled and visible, never hidden. On who held the phone in the eleven iterations, the project's files say both, the real user and me, and the chapter does not choose.
The decision-maker	The owner accepted the Lock in writing, tightened the speed budget himself, and kept the gate on fixed fees. The library's mirror is a deployment that an entitled person may commit and that is still refused over a fake.
The educator	The record supports only a small thing: the first application taught its user by doing, with guides whose words leave no doubt. There is no educator in this story, and I say so.
The agent	An AI agent can drive a production deployment and change nothing: the gate passes the services, nothing is committed, and the reason given is the actor and not the services. In the project, the AI agent is the session working beside me under the Lock, with a person deciding.

9. EXERCISE

Declare your own Lock for one cycle. Write the worlds of the cycle you are in, three of them, as a restaurant would: the customer's phone, the kitchen screen, and one more, each with the written reason it is in. Add a fourth, the idea you had this morning. Run the court. Change the declaration, not the court, until it says the cycle is sound.

HARO

```
? YourGraph.RulesAreSound()
#--> 1
```

✎ not run: an exercise, written for you to run

The court will tell you the circumstance and the rule. It will not write the declaration. If you want a harder one, add a rule about the actions of your cycle, and find the action that costs a third gesture.

I left this one in your hands, on purpose. The owner's speed protocol is at the seventy-fifth percentile, and the budget of section 3 is judged at the ninety-fifth. Take that block, change the percentile of one expectation to another that the library offers, keep every other line, and run it. Read what the court judges against what he asked for. Section 6 says what the library's terms offer, and section 7 lists his expectation among the things absent. I give you neither the verdict nor the answer.

When you have solved it, you will know why a promise the court can judge is not yet the promise the owner asked for.

Chapter 29 is a bank in Niamey, where an organisation is judged by its own chart, under a regulator's rules, by a program that reads the chart as data. The chart says what is, the rule says what must be, and the verdict names the rule and the place.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-service-virtualization-code-first-subscribe-later.md`
NARRATION	`stz-perf-judgment-narration.md`
NARRATION	`stzApp-building-your-first-app-narration.md`
ON THE SITE	forged
ON THE SITE	customers
RULINGS OF THE STUDY	12.15, 12.19, 12.25, 12.29, 12.30

PART VI · CHAPTER 29

A bank in Niamey

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 6 of 6 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take a bank in Niamey, which I call Softibank (the name is fictional), under the central bank of the West African monetary union, and I say first where the lesson is: in the library, and not inside the product the bank runs. I write the chart of a small bank as a text and judge it under the regional rule and under segregation of duties. I print what each of the seven regimes holds, add a rule of the bank's own as data, and read the one report with its verdict. I say the four things I refused, the first of them to hide the structure in a database. The principle is one sentence: the chart says what is, the rule says what must be, and the verdict names the rule and the place.

1. THE CONCEPT, AS THE FIELD HOLDS IT

A bank is judged on its structure as much as on its numbers. A regulator asks who reports to whom: whether the audit function is independent of the executives it audits, whether there is a board, whether a risk function exists. It asks whether the person who operates a payment is the one who approves it. The field answers with documents: an organisation chart drawn in a slide, a policy written in a word processor, an audit that compares the two once a year.

The field also has a family of tools called governance, risk and compliance platforms, which keep registers of risks and controls and track findings through to closure. They record that a control exists and who owns it. They usually do not

read the structure of the organisation and tell you that it is wrong. That is general knowledge, not a study of the record.

2. THE FIRST PRINCIPLE

An organisation can be judged by its own chart, under a regulator's rules, by a program that reads the chart as data: the chart says what is, the rule says what must be, and the verdict names the rule and the place.

3. THE SOFTANZA FORM

A commercial bank in Niamey, under the rules of the central bank of the West African monetary union, runs an organisation-assessment product I built. It is installed on the bank's own internal network, under licence, and the installation guide I delivered to the bank is the document that attests the reference. The product itself is a Python application: a small web server with a database. I say that first, because the lesson I am going to show you is in the library, and not inside that product.

The lesson is that an organisation chart is data a program can judge. It begins with a text that a person can edit, with no programming: positions, their titles, their departments, and who each one reports to. This is the chart of a small bank. What does the regional regulator's rule say of it? Run it:

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
}
```

✓ ran: printed what this block promises

The regulator's rule reads the chart and refuses it! Audit reports to the chief executive, and the verdict says so in its own words, `BCEAO-002: Audit reports to non-board position`, with a rule code an auditor can look up. A chart is judged by more than one court at a time, each finding carries its own code, and the compliance report says compliant or not. What does a second court say, the segregation of duties, which holds that operations must not report to treasury? I add treasury and operations, operations under treasury, and run both courts:

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
}
```

✓ ran: printed what this block promises

Two validators fail, each with its own code, and the compliance report says `non-compliant`.

The library names seven regimes, by class name: the regional one, Basel III, SOX, PCI DSS, ISO 27001, GDPR and HIPAA. They share one mechanism. How many rules does each hold? I print it, rather than write a sentence that says "Basel III is validated":

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
```

✓ ran: printed what this block promises

Six regimes carry four rules, and one carries five! Four rules every base carries: nobody reports to themselves, no reporting cycle, no orphan position, and a span of control. SOX adds one of its own: no position holds both the approver and the executor role. It is also called the separation of duties, but it is a different rule from

the second court above. The library's comment says it plainly: the regime classes differ by name today, and only one adds a rule of its own. The four are the rule base's; the validator that refused the chart in the first block holds three rules of its own, coded BCEAO-001 to BCEAO-003, and the two are not the same thing. I will come back to what that means.

A bank's own rule is data too. Take the rule that internal audit reports to the board. It is an object declared by calls. It names the department it governs, the line it requires and the sentence it prints, and it is added to a regime's rule base. Which rule broke, and where? The report says:

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
```

✓ ran: printed what this block promises

The report says `UNSOUND` and names the rule and the place: `ERROR audit-reports-to-board @ audit -- internal audit must report to the board`.

Then the verdict follows the chart. Two edits to two reporting lines, audit now to the board and operations to the chief executive. What do the courts that said no say now? Run it:

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
}
```

✓ ran: printed what this block promises

They say `compliant`, and the report says `SOUND`. That is the move the narration of a virtual bank tells, from a non-compliant structure to a governed one. The narration's own code does not run at the pinned commit; the calls above do. The report is one gate over the rule bases it is given. And here is the limit of it, as a block. A court reads what the chart declares and no more. The regional rule compares the department of a position's superior only when that department is declared, so I expect the finding on audit to vanish when the chief executive's department is left blank. What does it say when it is? Run it:

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
}
```

✓ ran: printed what this block promises

It says `1`: an audit function that reports to the chief executive passes! Declare the department, and the finding appears. The same block shows the banking validator answering `pass`: the library's own comment says it passes every time, a placeholder for rules not written yet.

My earlier book, *Refinement-Oriented Programming*, tells the discipline of governing change through the story of Softibank, a bank facing a regulator's letter. This chapter does not retell it. It tells what the library carries.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused to hide the structure in a database. In the delivered product, an organisation chart is rows in two tables. In the library, it is a text that a person can read and edit, and it imports into the object the court judges. A rule applied to a database row is a query; a rule applied to a text is something a person can read before it runs.

I refused to claim more than the library holds. The word "validator" can be read for two different things. The regional regulator has a validator of three rules, and a rule base of four universal ones. Basel III has the rule base only, and so do the others. I printed the count, and the block above shows it.

I refused to carry a regulator's text in a code I cannot source. The codes BCEAO-001 to BCEAO-003 are the library's own. The record ties them to no article of the central bank's text. They are the library's own rules, named after the regulator's regime; the record does not name a text they are drawn from.

I refused to tell what I do not know. What the bank's own people declare, operate or say is not in the record. The installation guide is my document to the bank: installation steps, a licence file, a support contact. The Forged page keeps no figure and no person of a client. The record is silent on who at the bank declares the chart, the questionnaire or the rules. It is silent on how long the product has been installed, and on how many people use it. I will invent none of it, and the chapter does not fill the gap. On "built with Softanza", the record shows a Python application, a small web server with a database, that calls nothing of the library. The library's validators are separate from it, and they are what this chapter runs. Chapter 1 says that building the platform shaped Softanza; this chapter says what the record shows.

Six guards stand behind the blocks.

5. THE COURT

`orgrule_narrated` proves SOX, GDPR and HIPAA as rule bases over the chart's graph, with the four universal rules and the separation of duties that SOX adds, and the chart joining the one report. The block above prints the seven counts. `14_bceao_governance` proves two of the three rules of the regional validator. `15_multiple_validators` proves the combined verdict, and `13_segregation_of_duties` the second court. `rulereport_narrated` proves the one report over findings from code, AI agents and security. `36_import_from_stzorg` proves the text form of the chart.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in coverage. A governance and compliance platform usually holds a register of regulations, controls and findings, with workflows, owners and deadlines. That is general knowledge, not a study of the record. The library holds one structural check of a regional regulator, one of segregation of duties, and a mechanism. For GDPR, HIPAA and the others, it holds the mechanism and the four universal rules, and nothing that is specific to them.

The field leads in actual use: such a platform is often one that a bank's auditors already know, and that a regulator has already seen. That is general knowledge, not a study of the record.

Where I think this stands apart is that the chart is the data and the court is a rule over it. A chart is a text, a rule is data, a finding names the rule and the position, and the report is one verdict. The library's own comparison table for this feature is all wins, undated, with cost figures that cite nothing, and I will not print it.

Now the limit, in the stage words.

7. THE LIMIT

BUILT	the organisation chart as a text a person can edit; the validators of the regional regulator and of segregation of duties; the seven regimes as one mechanism; a rule written as data; one report with one verdict
DESIGNED	the content that would make the other six regimes differ from the first; a rule file that judges, and not only loads
ABSENT	the delivered product running on the library; a word from the bank's own people; an auditor's reading of a finding the library produced

What is built: the organisation chart as a model and as text; the validators of the regional regulator and of segregation of duties; the seven regimes as one mechanism with the SOX exemplar. Built too: a rule written as data, and the one report with its verdict.

Four things are designed, or built in part.

- The content of the six regimes other than the regional one. Five of them hold the same four universal rules and nothing else; SOX adds a fifth.
- A rule file that judges. The library can load a file of rules in a `when` and `then` text, and the one in its test data loads two rules. The loader reads a rule's name, type, severity, function, params and message, and no `when` or `then` line. Such a rule loads and judges nothing. The format is admitted, and the rule is not enforced.
- A validator reads declared properties. It finds the board by the word "board" in a position title, and it compares departments only where they are declared: the limit block of section 3 blanks the chief executive's and the finding on audit disappears.
- The succession validator flags every filled position, by the library's own comment a known defect. The reorganisation simulation raises when it is asked to change a reporting line, also a known defect. The article that tells the simulation names a method that exists nowhere in the library.

Four things are absent.

- The delivered product running on the library. It is a Python application; a search of its code found no regional or Basel check and no call to the library.
- A word from the bank. The record is silent.
- A perceived verdict. No document says that an auditor or a supervisor read a finding that the library produced.
- A delivery date. The delivered files carry file-system times, and those are not a delivery date.

The web layer of the product was born in its own repository: its first commit is dated 4 February 2026, and the interface constitution's text was first committed on 8 February. The constitution's own evidence fields say that five of its rules were harvested from a banking-supervision product. The record does not say that product is the bank's installation. I will not say that it is.

Six doors follow, one for each reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	The chart is a text, the rule is data, the court is a call. A finding names the rule and the position.
The analyst	You change a reporting line in the object, or in the text and import again, and you re-run: the verdict follows the chart.
The designer	The report is one verdict over the rule bases it is given: sound or unsound, with each finding where it was found.
The decision-maker	Ask which of the regulations the library holds, and ask for the count. Seven regimes, one with a validator beside its rule base and one that adds a rule; the rest carry the universal four.
The educator	The narration of a virtual bank tells the move from a non-compliant structure to a governed one, and the book shows the move with calls that run.
The agent	You may read a chart and a report. A rule you add is data a person can read, and the court judges the chart with it.

9. EXERCISE

Draw the chart of an organisation you know as a text of positions, titles, departments and reporting lines. Before you run anything, write what you expect the regional rule, then segregation of duties, to say of it. Then write a rule of your own as data, one that governs one department, and say what the report will print.

Run it. Then leave one department blank and say which finding disappears, and why that is dangerous.

I left this one in your hands, on purpose. Take the limit block of section 3 and change one line: the import takes the chart whole, with the chief executive's department declared.

HARO

```
oOrg2 = new stzOrgChart("bank")
oOrg2.ImportStzOrg(cChart)
? oOrg2.IsValidXT(:BCEAO)
```

```
? oOrg2.ValidateXT(:Banking)[:status]
oOrg2.SetPositionDepartment("ceo", "executive")
? oOrg2.IsValidXT(:BCEAO)
```

✎ not run: an exercise, written for you to run

Run it and put the three answers side by side. The regional court has already refused this chart, in section 3; now read what the second validator says of it, and why it is not the same as what the court said. Section 3 and section 4 say why. I do not give you the answer here. When you have it, you will know why a validator that answers `pass` has not judged the chart.

The next chapter leaves the bank for an organisation in the field, and its principle is one sentence: the AI agent proposes, and a person decides.

WHERE THE EVIDENCE LIVES

NARRATION	`stzOrgChartDoc-BankUseCase.md`
NARRATION	`stz-rules-as-graph-rules-narration.md`
ON THE SITE	forged
ON THE SITE	customers
RULINGS OF THE STUDY	12.19, 12.29, 12.30

PART VI · CHAPTER 30

An organisation in the field

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 7 of 7 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I take one organisation's design study and show the four things in it that the library carries. They are a rule as an object whose numbers can be changed, a refusal a person can read, authority set against risk, and a decision kept with its authority at the time. Then I show the one thing the organisation asked for that the library does not carry, offline, and I read that block as a limit and not as a win. I say "design study" and not "delivery" every time, because the hub is not built. The principle is the organisation's own: the AI agent proposes, and a person decides.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field has two practices that this project needed, and keeps them in separate places.

The first is the business rule that a business person can change. Rule engines and decision tables have existed for decades; the Decision Model and Notation, a standard of 2015, gives business analysts a table in which an approval ceiling is a number in a cell. The second is the application that keeps working with no network. Offline-first design is a practice of its own, with synchronising databases. Since an essay of 2019 it has a name for its philosophy, local-first software: the data is on your device, and the network is a convenience.

A third practice has come with AI agents: an AI model proposes, and a person approves. In the field it is a button.

The organisation's study turns that third practice into a principle.

2. THE FIRST PRINCIPLE

The organisation's own study of the project gives it as the title of a section, and I take it as the principle of this chapter, in my translation of its French title: **the AI agent proposes, and a person decides.**

3. THE SOFTANZA FORM

An organisation works in Niger, with a head office in Niamey and bases in the field. Its own specification preferred a low-code platform built on a large vendor's office suite. My offer of 5 August 2026 was the one retained. The study lists four gains, and ownership is the first it lists. The organisation owns its vocabulary and its rules; it changes them without programming; it pays no licence per seat for the platform; and it can hand everything to a programmer of its own country. We built the design study together, with its teams, from 22 September to 8 October 2026. The hub is not built. The study's version 1.2 is submitted to the executive director for validation, and the record does not say it was validated.

Four things in that study the library carries. I will show them in the order the organisation would meet them, and then the one it asked for that the library does not carry.

A rule is an object whose numbers can be changed. An approval ceiling is a sentence with a number in it, attached to the model of the advances it governs. The model judges itself, and says how many subjects the rule looks at. The data below are invented for this chapter, four advances with an amount and a count of approvals. They are not the organisation's figures, and the numbers five hundred and two hundred are mine, not its thresholds. Does a ceiling of five hundred catch an advance, and how many advances does the rule look at?

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
```

✓ ran: printed what this block promises

The report names one advance, `a2`, at 600 with a single approval, and the rule looks at 2 of the four, `a2` and `a3`.

What happens when the ceiling is changed and the same model is judged again? Here the change is the rule rebuilt in code with another number, and the verdict and its scope follow:

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
```

✓ ran: printed what this block promises

The report now says `2 finding(s), 2 error(s)` where it said one, and the scope is 3 where it was 2! I have to be honest about what this block shows. That a change of the rule is a change of its values, which the organisation's own person makes like filling in a form, is the study's promise. I did not find a file of the library that makes it so. Section 7 says what does and does not exist.

What the library does carry is the answer to the worst thing that can go wrong when a non-programmer edits a rule: a rule that quietly governs nothing. I hold that as a view, not a result. Misspell a property, and the rule finds nothing, and the model reports that it is sound. What does a rule governor say of the same rule? The class is `StzRuleGovernance`; it asks rules what they govern and not what they say, and it names the dead rule:

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
}
```

✓ ran: printed what this block promises

The misspelt rule finds nothing, `0`, and to the model nothing found means sound! The governor answers `scope_empty`, and says why: `governs nothing in 1 pictures -- a dead rule and a misspelt scope predicate both report as passing`. A dead rule and an all-clear produce identical output: silence. The court has to tell "nothing was wrong" from "nothing was looked at". The word "pictures" in the message is the rule governor's origin: it was built for the graph plane's drawings first, and I would say "cases".

A person can read the refusal. When an answer breaks a law the organisation declared, it is refused with its reason. The refusal is kept as a checkpoint a person can read, with its context. Chapter 17 showed the asking half of this machine. Here is the refusing half. A request needs one approval, and the law says a request has one approver. What does the machine do with a second approver?

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
}
```

✓ ran: printed what this block promises

The machine asks first: `What does 'request-7' have for 'approved-by'? (why: 'request-7' needs 'approved-by')`. Then the second approver is refused! The checkpoint keeps what was attempted, `approver-2`, and the reason, in the refusal's own words: `'request-7' already bears 'approved-by' (to 'approver-1'); the relation is :Unique`.

Authority is not permission. The assistant of this organisation may prepare a file. It may hold the permission to release an advance and still be refused, because its authority does not cover the act's risk. A person with the authority is allowed. May the assistant release the advance, and may the approver? The verdict says why, in either case:

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
}
```

✓ ran: printed what this block promises

The assistant holds the permission for `release-advance`, and the answer is `0`! The verdict reads `refused: 'assistant' holds 'advisory' authority (level 1) but 'release-advance' is risk tier 3 (SHOULD does not cover it)`. For the approver it reads `allowed: permission held AND 'autonomous' authority (level 3) covers risk tier 3`. In the message, SHOULD is the library's word for what an authority level covers, as against CAN, the permission held; the guard behind this block says it as "can is not should".

The class here, `stzGovernance`, judges authority; it is not the rule governor of the dead rule above. The library knows an actor only by a name and an authority level. Whether an actor is a person or an assistant is what its level is declared to be. Binding a level to a signed-in person is the work of the authentication plane, which I have not read for this chapter.

A decision keeps its authority at the time. Each decision records its reason, its actor, the act, the authority and the risk of that moment. A new decision supersedes the old one without erasing it, up to a capacity (512 decisions by default), and the governance counts what it dropped. Record two decisions on the same case an hour apart: which one does the lineage answer with, and does the first survive?

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
}
```

✓ ran: printed what this block promises

The lineage answers `ceiling restored, the season is over`. It keeps 2 decisions, and the first still reads `ceiling raised for the harvest season`. The authority at the time is `autonomous`. The instant is a fixed number, so the clock is not read. This makes the block repeatable. It is not a version store for the rule itself, and I will not pretend it is.

And the one it asked for that the library does not carry. Its default option is to work offline: bases in the field with no network, entering what they do and sending it later. The library knows offline as a declared, governed capability of a world, asked for and granted or refused like the camera or payments. Is offline granted before the world asks for it, and after? That is all the library knows of it:

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
}
```

✓ ran: printed what this block promises

Before the request, `Granted(:offline)` is `0`, and the printed reason is `capability 'offline' was never requested via AddCapabilityQ().`. After the request it is `1`, and the printed reason is `allowed: permission held AND 'advisory' authority (level 1) covers risk tier 1`.

Read this block as a limit and not as a win. Nothing in it queues anything. The library does not keep entries on a device, send them when the network comes back, or settle two people's edits.

The next section sets down the decision, and what I refused.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused a per-seat platform as the organisation's ground. The specification preferred low-code on a vendor's suite. The retained offer proposes Softanza, and among its gains the study lists ownership first. The organisation owns its vocabulary and its rules, which it can change without programming, and the whole of the work, programs, configuration and data, is its own, with a handover to a local programmer. The vendor's suite stays where it was good, for the user accounts, the documents and the notifications. The record holds the outcome and the gains, and not the deliberation. Who chose, and what was weighed against low-code, it does not say. The chapter does not fill the gap.

I refused to let the assistant decide. It proposes. Its authority is advisory; a person's covers the risk. The refusal names the level, the risk tier, and the sentence that says the authority does not cover the act: `SHOULD does not cover it`.

I refused a silent rule. A rule that governs nothing is reported. A rule with no message is reported too. That is the price of letting a non-programmer touch a rule. The rule governor exists for that case, and the library's own comment says why: a dead rule and a misspelt predicate produce identical output, silence.

I refused to call offline a lesson. The Forged page, read on 9 October 2026, says: "This is a gap, not a lesson learned." A restaurant asked for the same. The study is clear that offline is the organisation's default, and the library has only a permission for it. A sister package of the estate, a browser-side one, holds a durable outbox on the device and an ordered, idempotent replay; it is outside the library. I did not run it.

Six guards stand behind the blocks.

5. THE COURT

`graphruleset_narrated` proves a rule with a comparison and a requirement turned into an implication. `gg_rule_governance` proves that a rule that governs nothing is reported. `wisecoding_narrated`, its third scene, proves a refusal that is governed and checkpointed. `governance_narrated` proves that can is not should: permission held and authority lacking is a refusal. `governance_lineage_narrated` proves the decision kept with its authority at the time. `01_platform_narrated` proves the capability seam: the camera granted, payments refused, a capability never asked for refused with an honest reason. The guard does not itself use the capability for offline; the block above does, with the tier the platform declares for it. The refusal of an unknown rule type, of a missing file and of a duplicate name rests on scenes 4 and 5 of `rulz_roundtrip_narrated`. That is a narrated test of the library, and this book does not run it as one of its guards.

The next section sets what the field does against what the library has.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in business rules for business people. Decision tables are a standard; a person edits a cell, not a program. Here the change is a rule rebuilt in code. The file form that would carry a rule as text, the `.stzrulz` file, reads a rule's name, its type, its severity, its function, its parameters and its message. It does not read the `when` and `then` lines of its own sample.

The field leads in offline. Local-first software is a practice with synchronising databases and a decade of experience. The library has a permission.

The field leads in people. A decision table is a thing a trained analyst already knows how to read. And the organisation's own specification, which preferred the vendor's suite, had good reasons: its users already have accounts there.

The next section lists what is built, what is designed and what is absent.

7. THE LIMIT

BUILT	a rule built as an object on a model, its number changed and the model judged again; the rule governor that names a rule that governs nothing; a refusal kept as a checkpoint a person can read, with its context; authority set against risk; a decision kept with its authority at the time; offline as a capability a world must ask for
DESIGNED	the hub and its study; a rule language a local programmer can learn; the study's promises that a rule is checked before it applies and can be undone
ABSENT	the hub, built; a mechanism for work done offline, kept and sent later; a version history of a rule; a customer platform on the engine

What is built: a rule as an object, its number changed and the model judged again; the rule governor that names a dead rule; the checkpoint; authority against risk; the lineage; offline as a capability.

What is designed: the hub and its study, whose schedule is in the future; a rule language that a local programmer can learn; and the study's promises about rules. The language's sample in the study is written in a notation that is not the library's. The study promises that a rule is checked before it applies, that its effect is shown before it is applied, that it can be undone and that its history is kept. The library has only the pieces. They are the rule governor, shown above, and the refusals of an unknown rule type, a missing file and a duplicate name, which a narrated test of the library proves and this chapter does not show.

What is absent: a version history of a rule with an undo. The lineage records the decision, and the rule is rebound. I found no store of rule versions, and my search was not exhaustive. The hub, built. Offline as a mechanism: no class in the library

keeps entries on a device and sends them when the network returns. And a customer platform on the Softanza engine.

Six doors follow, one for each reader.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	A rule is an object with a comparison and a requirement, attached to a model, and a rule governor tells you when it governs nothing.
The analyst	The study promises that you change a rule's values, not its program. In the library, when the values change, the model is judged again with the new scope.
The designer	A refusal keeps its context for the person who will read it.
The decision-maker	Authority is not permission: ask which level covers which risk, and read the lineage of a decision.
The educator	An organisation's rule can be taught as a sentence with a number in it.
The agent	You prepare. A person decides. And you will be refused, with the level and the risk tier named.

9. EXERCISE

Pick one rule of an organisation you know that has a number in it, a ceiling, a delay, a count of approvals. Write it as a rule with a condition and a requirement, on four invented cases, two that satisfy it and two that do not. Before you run anything, write the report you expect. Then misspell one property of the condition on purpose, and say what the model reports, and what the governor reports.

Run it. The difference between those two answers is the chapter.

I left this one in your hands. The block below is the second block of section 3 followed by the last lines of the governor block, with one change: the rule has no `SetMessageQ` line, so it carries no 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())
```

✎ not run: an exercise, written for you to run

Run it on the model of section 3. Compare what the model reports with what it reported when the rule had its message, and read what the governor returns. Under "I refused a silent rule", section 4 says what the governor does with such a rule, and why. I do not give the answer here. When you have solved it, you will know why the governor reports a rule that has no message.

The next chapter says what the projects show, and what comes next. Its measure is this: a project shows exactly what a document, a file, a run and a named person reach, and the page says where they stop.

WHERE THE EVIDENCE LIVES

NARRATION	stz-rules-as-graph-rules-narration.md
NARRATION	stz-governance-lineage-narration.md
ON THE SITE	forged
ON THE SITE	customers
RULINGS OF THE STUDY	12.13, 12.29

PART VI · CHAPTER 31

What the projects show, and what comes next

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 5 of 5 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I say, project by project, what the three projects of chapters 28 to 30 show today, and what each one makes the next step. I start with the field's three habits for saying what a piece of work has shown. Then I run five small blocks in which the library shows, by refusing, where a stage ends, and I lay out the tables: for each project, the claim, the document or file that backs it, and what comes next. I say what I refused, my memory as evidence first of all, and I give the court, the field and the limit in the stage words. The principle is the chapter's own sentence: a project shows exactly what a document, a file, a run and a named person reach, and the page says where they stop.

1. THE CONCEPT, AS THE FIELD HOLDS IT

The field has three habits for saying what a piece of work has shown.

The first is the reference: a logo on a home page, a customer story of two pages, the sentence "trusted by". It shows that someone paid or signed. It does not say what they ran, for how long, or what they said.

The second is the readiness scale. NASA's technology readiness levels, later an international standard, name how far a technology has travelled from a laboratory to the place it was built for. The highest level is a system proven in operation. It is a vocabulary of stage, and this book uses one of its own (built, designed, absent).

The third is the limits section. An empirical study must state its threats to validity. A machine-learning model ships with a card whose last part is its caveats. An artefact submitted to a research conference is badged by what someone else did with it. This is the form closest to this chapter, and the one most often skipped when the reader is a buyer.

What the three have in common is that the claim and its proof travel apart. A page says "used by", and the proof is a trust in the page. That is general knowledge, not a study of the record.

2. THE FIRST PRINCIPLE

The record gives two sentences, and they are one. The site says: "Nothing is a reference until a document proves it." And, of what I remember: what "no file proves is not listed."

A project shows exactly what a document, a file, a run and a named person reach, and the page says where they stop. This is the chapter's own sentence, drawn from the site's two. The site's words are the two quoted above, from its customers page and its forged page.

3. THE SOFTANZA FORM

This chapter is the other half of chapters 28 to 30. Those told what three projects taught the library. This one says, project by project, what each project shows today and what it makes the next step. Parts of Softanza are in production, and others are in development. I did not write this chapter after the fact, to be modest: the site's forged page already carries a section on the same subject.

The shape of the answer is a table of three columns for each project: the claim, the document or file that backs it, and what comes next. I leave a cell empty when I cannot name what fills it. And I will not fill the third column from memory.

Before the tables, five small blocks. Each is a place where the library shows, by refusing, where a stage ends, instead of passing silently. They are the chapter's sentence, written as code.

A dependency declared and never bound is an error. The surface of a solution is declared before anything is bound. A declared service that nothing backs is counted as a finding, not a pass, and asking for it raises where the mistake is. This is the first step the restaurant's lesson points to: binding a real payments service, which the record does not yet hold. What does the registry answer when the payments service is asked for and nothing backs it? Run it:

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.
```

✓ ran: printed what this block promises

Two services declared, none bound, and the registry says so itself: `nothing is bound for service 'payments'!`

What a rehearsal can claim, and what only the target can. The platform model gives three verdicts for the same operation. *Refused*: the target forbids it. *Native*: the host itself can do it. *Rehearsed*: the target allows it and the host lacks it, so the run shows only the plan. Reading a pin on a development machine is rehearsed, not native. The word depends on the machine this book was run on. A Windows PC has no such pin, and a board that has one would print `native` for the same line. What does the board's scope answer to a pin read, and to a request to spawn a process? Run it:

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
```

✓ ran: printed what this block promises

this block touches the system; the output shown was recorded on 2026-10-15 12:58 UTC+01:00

The pin read answers `rehearsed`, and the scope refuses the process with `forbids 'process'`!

The library's own comment, in the class that lowers a rehearsal to a real artefact, names the next step: the last one, flashing a board, "remains a PLANNED external action".

The live payments adapter exists, and refuses to guess. The code for the live payments system exists. Its run against the organisation's sandbox has not happened yet, and it is the next step. Given no credentials, it refuses before any network call and says what is missing. What does it say when it is asked for accounts and holds no credentials? The host in this block does not exist, and nothing reaches it:

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
```

✓ ran: printed what this block promises

The adapter answers `no credentials`, names what would fill the gap, `WithSecretsFrom(oStore, oActor)`, and `TokensFetched()` stays at `0`.

The library has a script that tries the live run. I ran it once, with none of its variables set, for this chapter. It printed the word UNPERCEIVED, named the four variables that are missing, and said that nothing about the live adapter is perceived until a named person has watched a payment land. It ended without failure. The run was postponed, by me, for a reason that is not the library's: it needs the organisation's involvement. A run that could not happen is reported as unperceived. It is not reported as a pass or a failure.

A feature that waits for a named person. In the sound plane, the library holds as data the verdict of a person on a thing only a person can judge. While the verdict is UNPERCEIVED, the verb that would claim the result refuses and says why. Where a person has said yes, the verdict carries the date and their words. What does the formant voice answer when it is asked to sing, and what does the retuned voice's verdict hold? Run it:

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()
```

✓ ran: printed what this block promises

The formant voice refused to sing: `Refusals()` reads `1`, and the message begins `Sing is not available YET`! The verdict of the formant voice stays `UNPERCEIVED`. The verdict of the retuned voice reads `SINGING`, with the date `2026-09-27` and the words `the retuned voice is somehow singing, open Sing()`. This is the form of the line that chapter 33 sets as the next milestone for a customer's run: a date, a person, and a sentence. The record also carries the person's name beside it, and I do not print it here. SAPI, in the message, is the operating system's speech interface.

A pre-flight that says "feasible" without touching a host. The deployment's pre-flight refuses a host that is too small. For a host whose provider is a cloud, it says feasible and "provision on aws", without any account. A feasibility is a plan, not a verification. The design file names the next step plainly: the verification against a real host or account, which waits on infrastructure. The figures below are illustrative. What does the pre-flight answer to a host that is too small, and to a cloud provider with no account? Run the block:

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).SetStorag
eQ(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
```

✓ ran: printed what this block promises

The tiny host gets `SHORTFALL -- host too small`, and the cloud with no account is feasible, with `provision on aws`!

Now the tables. Each one opens with what the project delivered or ran, and ends with what comes next. Every cell is from the record, read on 8 October 2026. A line count or a file date I read myself is labelled as mine.

THE RESTAURANT IN LYON

CLAIM	PROOF	WHAT COMES NEXT
The platform exists, and its first delivery was made	the site: an amendment closed on 4 August 2026; the project's commit "iteration 1" of 15 August; a server of 565 lines that I read; the last commit, 25 August 2026, 13:07	25 August 2026 is the last dated record; the next record is the one the next phase writes: production, a real order, the planned go-live gate
The first iteration was	the site: "first iteration delivered"; the project's article: a day of field	the owner's own sentence on how he uses it is the next record to gather; the record's

CLAIM	PROOF	WHAT COMES NEXT
delivered, and a field day followed	measurements with two phones, 19 August	words are "first iteration delivered" (the site) and "a day of field-testing" (the project's article)
No growth of scope during a cycle (the Lock, chapter 28)	the pact and the notebook of the Lock: private documents	the Lock as code; chapter 28 shows the nearest the library gives
Payments can be tried without a subscription, and a fake never ships	the registry and the production gate, with their guards	the live payment adapter for the restaurant, "a later desk" in the library's own design file; in the library's registry guard a sandbox is bound, and the live binding is the step after it; the restaurant's own solution does not call the registry
The constellation of worlds	the restaurant's map of its worlds; the library's super-app class, with a guard	the restaurant on the constellation: its worlds are single-file web pages today, and the map says it is a map, not an implementation
A speed budget is a promise	the library's service-level object, with a guard	a measure on a real 4G network: the project's dashboard of 23 August moves it to the next cycle
The interface rules were forged with the owner, phone in hand	the project's file says eleven iterations with the real user	who held the phone: the record does not hold it, and my study of the record ran the two facts together

THE BANK IN NIAMEY

CLAIM	PROOF	WHAT COMES NEXT
The bank has my assessment product, installed on its own internal network under licence	the installation guide delivered to the bank; a licence file	the bank's own account: who declares the chart, for how long, how many people use it. And a date of delivery: the files carry file-system times, which are not dates
The product is built with Softanza	I built both	the product calling the library. Today it is a Python application with a small web server; I searched its code, and found no regional or Basel check and no call to the library

CLAIM	PROOF	WHAT COMES NEXT
The regulator's rules judge the chart	the library's validator of three rules and of segregation of duties, with guards	the validators inside the product; an article of the central bank's text for the three rule codes; a finding read by an auditor
A rule file for the bank	a file of two regional rules that loads	its first judgment: on a chart with no board it reports sound today
The product's interface rules came from the bank	five rules of the interface constitution	the evidence line of the first of them reads "A banking supervision product (April 2026)" and names no bank. My study of the record attaches them to this bank, and the record does not

THE ORGANISATION IN THE FIELD

CLAIM	PROOF	WHAT COMES NEXT
The organisation chose my offer	the offer of 5 August 2026 was retained; the design study, built with its teams from 22 September to 8 October 2026	the organisation's hub, not yet built; the study's version 1.2 is submitted for validation. And who chose: the record is silent
Rules are data that a person changes without a programmer	the library's rule clauses and the governor that names a dead rule	a text form for clause rules; a version record with undo, which the organisation's study promises; its own rule notation is not the library's
An AI agent prepares, a person decides, with the authority at the time	the governance object and its lineage, with guards	a person bound to a signed-in account; a hub to do it in
Offline by default	the organisation's study's word that a browser-side outbox is already in service; it is a sister package outside the library, which chapter 30 names and I did not run	the mechanism in the library is to come: no queue, outbox or synchronisation yet. The site calls it "a gap, not a lesson learned". Chapter 30's last block is the whole of it

ACROSS THE THREE

CLAIM	PROOF	WHAT COMES NEXT
The platforms run today on ordinary stacks	the site's own page: single-file web worlds, a Python application, the organisation's hub not built; and a small Node server that I read	the first platform on the engine. The site says so, and every repository I opened agrees
Their declarations "are being written as Softanza's own"	the platforms page and the forged page, as intent	the declarations as files are the next piece to write. I searched, on 8 October 2026, and did not find them as a file: in the restaurant's repository no code file declares an app, a constellation or a platform; in the bank's delivery, none; in the organisation's study folder, none (its annex uses a notation that is not the library's). The record shows the intent and the maps; the declarations are the step that follows
A change of engine, not of plan	the principle: rewriting was refused	the move itself; the record does not yet say who moves first or who perceives it
A solution writes its guarantee sheet in five rows	the vision asks for it	the first sheet is to come: no solution has one yet, and no class, fixture or guard in the library holds the sheet
The runtime beneath is the language's own	the charter ratified on 1 October 2026; its stages pass on its own fixtures	Haro running the library is the next step. A session's note of 5 October 2026 reports that the bridge runtime passes every assertion the old runtime passes on the library, though not every program agrees on every stable line: of 5,112 programs, 3,425 are identical on every stable line, 1,267 more differ only on the line that prints the elapsed time, and 19 differ in kind, as chapter 27 reads the note; the Linux pass is the step after that. I print no percentage. This book's blocks run on the former runtime, 1.27
A live host verifies a	the deployment design	the verification waits on infrastructure and is the next step. Local sites run for real; remote transfer is "a later slice"

CLAIM	PROOF	WHAT COMES NEXT
deployment		
The payments plane works against the organisation's sandbox	the twin, served over HTTP, with a guard	the sandbox run: postponed, unperceived so far. Production is never from this plane
A named person has said what they saw	a drum, and the retuned voice, in another plane	the same line for any of the three platforms, which is the next milestone

WHAT IS STILL TO BE MEASURED

My study of the record read it and did not run it, and it says so; here is what it leaves to measure next. Whether Haro's runtime runs the library is the first measure still to come. Whether two published speed benchmarks hold today is the second, and both were run on the former runtime. Some guards in the system folder took their totals from design documents, and one of those documents still says forty-five where its guard now holds more; the document is the next to update. The forty-one pictures of part 1 of the gallery were judged by an AI model, and a person looking at them is the next step. The restaurant's speed budget awaits its measure on a real 4G network. Each of these is a measure not yet taken, not a failure.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused my memory as evidence. The site lists nothing that I remember and no file proves. The same rule bound the bank: I know the product, because I built it, and what the bank's own people do with it is not yet in a file.

I refused to call the library the origin of what it only uses as an example. The forged page says "example" where the library holds a project as its example and not as its origin. And I will not repeat a sentence of my study of the record that says a lesson was "born of the client's need". The registry and the SLA were first committed after the reversal of 4 July, by the library's log as chapter 28 reads it. The order of the commits does not support "born of". What I can say is "needed" and "carries".

I keep the record's wording for what each customer runs. The record shows, for the bank, a Python application installed under licence and built beside the library by me, and, for the restaurant, a first delivery and a field day, with the record ending on 25 August. The wording that the record bears out is *built by me, under the discipline the library now carries*, for the bank, and, for the restaurant, *first iteration delivered* (the site) and *a day of field-testing* (the project's article). Chapter 1 says it in its own words, and this paragraph is where the record's more exact wording stands.

I refused a stage word the repository does not carry. The site's estate page says it: never called available before it is. A piece of work that reaches a person through a sense is perceived by a person. Until then it is unperceived, which is a state and not a failure.

I refused to print a percentage. I have said aloud how far the bridge runtime has come, as a share of the way, and that figure is not printed without the guard that produces it. The factor of section 6 is another number: a speed ratio measured in chapter 27, row by row, against a fast peer.

5. THE COURT

`service_registry_narrated` proves that a declared service nobody backs is an error finding, that an unbound service raises, and that a sandbox in production is a violation. `full_scope_model_narrated`, scenes 5 to 8, proves the three verdicts: refused, rehearsed, native. `payments_live_adapter_narrated` proves the adapter against a twin served over HTTP, an adapter without credentials that refuses to guess, and the scenario that says stage (b) is unperceived. It starts real server processes, and the book does not run it; it only counts its assertions. `sound_mu5_narrated` proves, in its fifth scene, that while the verdict is unperceived `Sing` returns nothing and says why. `deployment_narrated` proves that an undersized host is not feasible, and that an assistant rehearses and commits nothing; it writes a scratch folder beside itself, so the book does not run it either. `orgrule_narrated` proves the universal rules of the chart and the separation of duties. `14_bceao_governance`, one assertion, runs the regional rules on a chart with no board and no risk function, and two of the three fire, as chapter 29 says. `governance_narrated` proves authority against risk, with its lineage.

The court of this chapter's subject is not a guard. It is a line of the record, of the form the estate's law prescribes: a date, a person, and what they said. The library holds one such line in its data, in the sound plane. The estate holds one in its record, and chapter 33 tells it.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in production. Ordinary stacks run real workloads for real users. That is general knowledge, not a study of the record. The site's own page places the platforms of this part on ordinary stacks today, with the hub not built; moving

their declarations onto Softanza's own engine is the next step.

The field leads in references at scale. The site prints two references and a design study, and quotes no number of countries and no number of programmers. A buyer reading a logo wall sees more.

The field leads in live infrastructure. Provisioning tools drive real providers every day, which is general knowledge. The library's deployment reaches local sites for real today, and the real host is the next step; the fifth block shows where the line runs.

The field leads in hardware in the field. The declared machine boots and judges itself in an emulator, and the vision records that no customer is on it yet. Its guarantee text says four of four promised, three kept in the emulator, and carries no slot decision; the vision adds that the fourth, surviving a cut, waits for the watchdog on the board. The board is on order.

The field leads in speed. The runtime Softanza measured itself against was faster on every row, by a factor between one and two: 1.04 to 1.88 times, the six rows that chapter 27 prints from the memo of 13 September 2026. That factor is a measurement, not the progress percentage section 4 refuses. Both benchmarks were run on the former runtime; Haro's own speed is the measure still to come.

The field leads in pictures judged by people. Forty-one pictures and text sections of part 1 of the gallery were judged by an AI model: twenty right, nine wrong, twelve uncertain. The record says that no person has looked at them yet, and that look is the next step.

Where I think the record stands apart is that its limit is a section of a page, with a heading. Every stage has a word. A feature can be built and still refuse until a named person has spoken. And a live run that could not happen is reported unperceived, never a failure and never a pass.

7. THE LIMIT

BUILT what the three projects delivered, each with a document (a first delivery followed by a field day; a product installed under licence; a design study

built with the organisation's teams); the three lessons the library carries, each with a guard (a fake never ships; a chart judged under a regulator's rules; authority against risk, with its lineage); refusals that show an absence instead of hiding it

DESIGNED — the five-row guarantee sheet; a runtime that is the language's own running the library; the declared machine on a board

ABSENT — any customer platform on the engine; the customers' declarations written as Softanza's own, as files; a word from the bank; the record of the restaurant after 25 August 2026; the organisation's hub; an offline mechanism in the library; a live payments run against the organisation's sandbox; a named person's verdict on any of it

This whole chapter is a stage report. Here is its sum in the stage words.

What is built: what the three projects delivered, each with a document (a first delivery at the restaurant, followed by a field day; a product installed at the bank under licence; a design study built with the organisation's teams), and the three lessons they gave the library. *A fake never ships*: the registry and the production gate. *An organisation is judged by its own chart under a regulator's rules*: the organisation chart and its validators. *Authority is not permission, and a decision keeps its authority at the time*: the governance object. Each has a file of the library that carries it, a guard that resolves, and a block already kept in a neighbouring chapter. The next step of each is in the tables. In the library's registry guard a sandbox is bound, and the live payment adapter is the step after it; the bank's product does not call the chart validator. The organisation's hub is not yet built, and its offline mechanism is to come.

What is designed: the five-row guarantee sheet; the runtime that is the language's own, running the library; the declared machine on a board.

What is absent: any customer platform on the engine; the customers' declarations written as Softanza's own, as files; a word from the bank; the record of the restaurant after 25 August; the organisation's hub; an offline mechanism in the

library; a live payments run against the organisation's sandbox; a named person's verdict on any of it.

The sum of every next step is the line that chapter 33 sets as the next milestone: *the solution ran on Softanza's own engine, and this is what the user said*. The customers' declarations are its first piece, and they are absent today. The site says they are being written as Softanza's own. On 8 October 2026 I looked for a file of them in the restaurant's repository, in the bank's delivery and in the organisation's study folder, and found none. On 9 October 2026 I looked again, read-only, in thirteen repositories beside the library, and found no file that declares a customer's platform as Softanza's own. The chapter's word for them is *absent*, and the stage line at the top says so.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your demo is a rehearsal until a target has run it. Ask the scope whether the line was native, rehearsed or refused, and write the same word beside your own claims.
The analyst	When you cite a project, write three columns: the claim, the document, and what comes next. Do not fill the third from memory.
The designer	A feature that no person has seen or heard ships shut, and says so. The stage is a state of the design and not an apology. I hold this as a view, not a result. What stands behind it on this page is the sound plane, where `Sing` refuses until a person has heard the voice.
The decision-maker	Read the stage word before the testimonial. "Built" is a statement about a guard. "Runs at a customer" is a statement about a document. You can ask for it in writing, with a name beside it.
The educator	Teach the difference between a project that taught the library and a project that runs on it. The first is shown here, and the second is the next milestone, in chapter 33. Unperceived is a state, not a failure.
The agent	You may read the tables and quote any cell. You may not turn an absent cell into a present one. If you do, the next reader of the record will find your sentence and the missing file side by side.

9. EXERCISE

Declare the surface of one project of yours: a mail service, a payments service, and one more of your own. Before you run anything, write the line you expect under each of three questions: which services are unbound; whether the registry is sound; which are unbound once the mail service has a sandbox bound.

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
```

✎ not run: an exercise, written for you to run

Then, on paper, make the table of three columns for the project you know best: the claim, the document or file that backs it, and what comes next. Leave empty what you cannot name. On the last line write who perceived it, on what date, and what they said, or the word "unperceived" and why.

The exercise has no answer to be checked first. The question the tutor asks is: which cell did you fill from memory?

I left this one in your hands, on purpose. Below is the first block of this chapter with one line added: it binds the payments service to the library's sandbox before the registry is asked whether it is sound.

HARO

```
oReg = new stzServiceRegistry("shop")
oReg.DeclareMany([ :mail, :payments ])
oReg.Bind(:payments, new stzPaymentsSandbox())
? oReg.IsSound()
```

✎ not run: an exercise, written for you to run

Run it and read what `IsSound()` answers. Then ask whether a sound registry is a registry ready to ship, and what the guard of section 5 says about a fake still bound in production. I do not give the answer: section 5 names the guard that holds it, and section 7 says what the project has bound and what it has not. When you have solved it, you will know what a sound registry does not say about the service behind it.

Chapter 32 turns first to how many minds worked on this one body of work, and to the rule they keep: every law cites the incident that earned it, and a new law without a dated failure behind it is a preference. Chapter 33 then closes the book on the next milestone, a solution running on Softanza's own engine.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	customers
ON THE SITE	forged
ON THE SITE	platforms
ON THE SITE	estate
RULINGS OF THE STUDY	12.5, 12.10, 12.13, 12.21, 12.22, 12.29, 12.30

PART VII

What comes next

32 · Many minds on one body of work

33 · The next line: a solution on its own engine

PART VII · CHAPTER 32 · SPINE

Many minds on one body of work

DRAFT

Run on 2026-10-15 13:00 UTC+01:00 inside the library at commit 93d7a39bd: 3 of 3 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I show how Softanza has been written for the last two years: by many sessions of an AI model on one machine, with no framework, only files, a version history and a few hooks. I walk through the rules the record paid for, one failure at a time, and the library's fourteen house rules that an AI agent can run before it commits. I say what I refused: a framework, a coordinator that grades its legislator, automation to the maximum, a ruling in my absence, a protocol sold as a product. And I say the loss I accepted: my hand thinned where the sessions wrote. The principle is one of the sessions' own sentences: every law cites the incident that earned it.

1. THE CONCEPT, AS THE FIELD HOLDS IT

When the field speaks of many AI agents working on one body of code, it usually means a framework. A program holds several agents, a supervisor routes messages between them, and the coordination is a feature of that program. Microsoft's AutoGen, CrewAI and LangChain's LangGraph are the names you will meet, all from the last three years.

When the field speaks of the rules those agents follow, it means a file the agent reads when it starts, a page of instructions in the repository. And when it speaks of many writers on one repository, it means what teams have always used: branches, review, hooks that run before a commit, a list of who owns which folder. These are good tools, and I use them every day.

What I did instead begins with one sentence.

2. THE FIRST PRINCIPLE

The one sentence this part of the record returns to was written by one of the sessions, and I keep it because it is right: **every law cites the incident that earned it. A new law without a dated failure behind it is a preference, and belongs in a style guide instead.**

3. THE SOFTANZA FORM

Softanza was not written by one hand for long. For the last two years, most of its new code has been written by sessions of an AI model, many at a time, on one machine. That work spans some twenty repositories that depend on each other. I rule; they propose, build, measure and report. The record calls me the Principal. In this chapter I will say I.

There is no framework here. No supervisor, no broker, no graph of agents. The arrangement is files and a version history. One repository, the central one, holds the protocol: how a session reports, how it asks, how it closes. Each repository has a mailbox. Each session writes its messages in one fixed form, eight keys in a fixed order. The eight keys: by whom, about what, why, what was done, the state of named things, what is waiting and on whom, what comes next, and one line of judgement. Every message is also appended to a dated file, so that the day's reasoning is never only in a chat window that closes.

What makes it a discipline and not a filing system is that each rule was paid for.

Stage by explicit path. Several sessions write in the same repository at the same time. Twice, a session added every changed file to its commit at once, and swept in another session's work. Once it destroyed an edit outright, once it pushed half of a colleague's move between two repositories. Nothing was lost the second time, and the record says that was luck, not method. So now a hook refuses the command before it runs. It was tested on twenty forms of the command, and eighteen

behaved as written. Of the other two, `git stage -A` was repaired the same day; the other, that the hook also refuses text which only quotes the form, is an accepted cost, met in section 6.

Read the time from the clock, never compose it. A session writes a time on every message. Sessions, it turns out, invent times: they write a time that feels right after the last one. Nobody notices until a made-up time crosses midnight and a message is filed under the wrong day. How did the central session fare when it measured itself? Five of its twenty-five stamps had been composed! The worst was out by forty-two minutes, and four of the five were older than the law that forbade them. Across the other desks, thirty-four of a hundred and fifty-one stamps were composed, on thirteen of the seventeen desks measured, a desk being one repository with its mailbox. The rows behind that figure are published. The first time it was published, the rows had been thrown away, and the number could not be defended three hours later. Five weeks after the law of 25 August, the central session composed a stamp again. Four written warnings had not stopped it; a peer reading a file did. The note it left is the best sentence in the protocol: a reminder is not a mechanism.

Words name states, never people. A session once reported my working tree as "dirty". Work that is not yet committed is not dirty, and often it contains no error at all; and a word that judges the work judges the one who made it. So the record keeps a table of words the estate does not use, with the word it uses instead.

What no counter can see. What did the counters say of the sound plane of the library? Six hundred and twenty-two passing assertions across nineteen guards, and yet to any listener it was broken! I heard four defects that no counter reported: a voice too quick to understand, a sentence cut before its end. The law that came out of it says that a thing nobody has perceived is in a state, not in failure: unperceived is a state, and the page must say so. It is the same instinct as the stage every page of this book carries, Draft, until I have approved it.

The loop is seconds, not minutes. I stopped two sessions over the same eight-minute test run. It was measured: six seconds of the eight minutes were coverage, the rest were three sections re-rendering the world to test one pixel property. It

was rewritten to twenty seconds. I had written, in my own words, that it "takes a lot of time and should be rethinked". The law was written by a session; the sentence was mine.

Never ask without saying how. A session once asked me to check whether a figure was right, and gave me no way to check it. I answered: please never demand a thing without telling me how to get it. Now every request addressed to a person carries the exact command, the exact path, the exact line to read.

You can see the shape. A failure, dated. A rule, written where the next session will read it. Then, when the rule is broken anyway, a mechanism. Some laws reached the last step, some are still a written rule, and the record says which.

Where the laws touch code, the library itself can hold them. Its fourteen house rules are validators, five of them over a graph of the code and the rest over text: one law of the library's design document, made runnable. They are rules an agent can run before it commits, not folklore it must guess. One of them is about words. What does it say of a method called `KillOrder`, and is the code still clean? Run it:

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
```

✓ ran: printed what this block promises

Read the last line twice. `StzCodeIsClean` still answers `1`: the code is clean! The tone rule warns; it does not refuse. A word rule advises. The estate's doctrine says something close about its own word law: no session can test how a word lands on its reader, so that law is ruled rather than checked. The message says `aggressive`

`verb in method 'KillOrder'` and names the remedy, `prefer Remove/Delete/Dispose/Clear/Close`. What does the rule say once the verb is `Remove`? Run it:

HARO

```
cCode = "class Shop" + nl + "def RemoveOrder()" + nl + "  return" + nl
? len( StzCheckCode(cCode) )
#--> 0
```

✓ ran: printed what this block promises

No finding comes back: `0`. Once the verb is `Remove`, the rule has nothing to say.

And the rules are a closed set. How many are there, and can an agent list them before it writes?

HARO

```
? len( StzCodeRuleNames() )
#--> 14
? StzCodeRuleNames()[3]
#--> no-aggressive-verbs
```

✓ ran: printed what this block promises

Yes: fourteen names, and the third is the rule that just spoke. The record does not link the library's tone rule to the estate's word law, ruled on 21 August 2026, so I will not either. But it is the same instinct at two altitudes: the words we use about work are part of the work.

Those are the laws. Now for what I refused, and the loss I accepted.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused a framework. The arrangement is files, a history and a few hooks. A supervisor that schedules AI agents would be one more program to trust, and one more thing that decides without a record. Here, a session that is not running leaves only its files, and the files are the truth.

I refused to let a coordinator grade its legislator. Authority runs down from me; correction runs up, and the central session is corrected by the Observer, a desk that audits how the harness is built and repairs nothing itself, and bound by the doctrine, the written laws of the estate. A coordinator that judged its own rules would be the one part of the system nothing could fix. I hold this as a view, not a result.

I refused to automate to the maximum. From the middle of August 2026, sessions were opened by clocks. On 27 August I stopped them. The record keeps my reason as written then: the routines consumed quota without added value. And it keeps the part still open: that judgement was the only measurement anybody had, because the cost line every session writes at its close carries no token count. The harness does not tell a session what it spent. Of the three hundred and eighty cost lines written so far, one carries a number! What did the clocks cost? It is not in the record: it holds the reason and no figure, and I give none. Since then the ruling is that a timer may run a script, never an AI model, and every timer is paused for now. The direction is that a session starts when a file arrives in a watched folder, or when I start it; the watcher is designed and still to come (section 7).

I refused to rule in my absence. A ruling is written in the record only while I am there. On 28 September I asked the central session to settle what was pending on my behalf. The five rulings it took that day are marked as taken on my delegation. The mark is there so that nobody later mistakes them for mine. The decisions taken for this book on my delegation carry the same mark.

I refused to let the protocol become a product. The doctrine of the harness is open, and it is never to be sold: its value is in what it refuses. So far it has been proved in one place: one person, one machine, one filesystem, about fifteen sessions. The first adopter outside this estate is the next step.

And I accepted a loss. The code of the last two years was largely written by sessions, and my hand, which was the style, thinned where they wrote. My study of the record measured it. The promise under each line (the `#-->` line) is the house's way of writing an example. It stands at nearly a hundred per thousand lines in the short recipes (the quickers) and at seventy-five in my articles. In the narrated guards the sessions wrote, which assert in another form, it stands at under one. The sessions even took the name of my articles for their test files, a name that a model proposed. I accept the implementation code they wrote. I do not accept that the style a programmer or an AI agent writes with Softanza should be theirs rather than the house's. That is why the law of Part II exists, and why this book is written in my voice, with sessions as researchers and as the court. And it is why every chapter waits for my reading before it loses the word draft.

Next, the court, and what it cannot prove.

5. THE COURT

No guard of the library proves the protocol. Its courts are outside the library, and I name them by what they do. One hook refuses wholesale staging, and one refuses work when the machine has no memory to spare. A check says whether the hooks installed are the hooks written. An instrument compares every stamp with its commit and publishes the rows. They are scripts, not Haro, and this book's reader cannot count their assertions, so they are named here and not in the court line below.

The blocks of this chapter stand on the library's guards of its house rules. `coderule_narrated` proves that a code rule is a rule over the graph, and that the house findings come out on a planted fixture in a frozen shape. It proves that the

rules number fourteen, and that a definition or a comment is not taken for a call. `codegraph_narrated` proves the graph beneath them, and `coderule_project_narrated` runs the project rules over a fixture directory.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in scale and in independence. Its frameworks run on many machines for many teams; this protocol is proved so far at one person, one machine, one filesystem, and the first trial outside the estate is the next step. Its doctrine says so in its own words: these are laws that survived one estate.

The field leads in accounting. A paid interface reports the tokens of every call. This estate's cost record carries no token count yet; adding it is the next step, and until then the most important economic decision it took was taken by judgement.

The field leads in tooling for many writers. Branches, merge queues and owners of folders handle concurrency at a scale the estate has not met. Here, the answer is a discipline and one hook. The hook reads the text of a command, so it also refuses a commit message that merely quotes the forbidden form. That cost is known and accepted.

Where I think the record stands apart is in one habit: each law carries the failure that paid for it, and the record publishes its own rate of breaking its own law. The central desk broke the stamp law about as often as the estate it measured, on a different surface: 5 of 25 against 34 of 151. Four of the five came before the law. Both numbers are printed side by side.

Now the limit, in the stage words.

7. THE LIMIT

BUILT — the protocol, its memos and its ledger of rulings; three hooks: one refuses wholesale staging, one refuses a register edit made on a stale view of its source system, one refuses work on a flooded machine; the instrument that measures composed stamps; the library's fourteen house rules, five of them over a code graph

DESIGNED — a watcher that wakes a session when a file arrives, in place of the clocks; a court of style for the code a programmer or an AI agent writes

ABSENT — token figures in the cost record; any team outside this estate that has tried the protocol; a check of a word in the reader's own language

What is built: the protocol and its files, the dated memos, the ledger of rulings, three hooks that run in every session on this machine, and the instrument that measures stamps. The library's fourteen house rules are built, five of them over its code graph.

What is designed: a watcher that wakes a session when a file arrives, which, in my view, is the right replacement for the clocks and is still to come. And a court of style for the code that a programmer or an AI agent writes with the library. The house style would then be held by a machine as well as by my reading.

What is absent: a token figure in the cost record; any team outside this estate that has used the protocol; and a check that a word reads well in the reader's own language. The word law asks for that check, and no tool performs it yet; that check

is the next step. My study of the record named one more limit: the protocol that makes the record trustworthy also costs energy the product needs, and I should watch the ratio. The record holds no reading of that ratio.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Stage by explicit path, read a file again before you write it, read the time from the clock. Each rule cost someone an edit.
The analyst	The record is data you can query, and it publishes its own error rates beside its rules.
The designer	A word names a state, never a person. A message is a closed structure a reader scans in one glance, written so that it parses for a reader whose first language is not English.
The decision-maker	A ruling is written only in your presence. When no figure can answer a question, decide by judgement and write down that it was judgement.
The educator	Teach a rule with the failure that earned it. A rule whose cost you cannot name is a rule you will talk yourself out of at two in the morning.
The agent	You are a worker with a record. You write a memo, a cost line and a conclusion; the hooks refuse what has hurt others; your stamps are compared with your commits. None of it assumes you are careless. All of it assumes you are many.

9. EXERCISE

Take the last ten messages, notes or log lines in which you wrote a time by hand. For each, find the moment the system recorded on its own: the commit, the time the message was sent, the time the file was saved. Before you look, write down how many of your times you expect to be later than the system's by a minute or more. Then count.

Your count is the finding. I will not give you mine.

I left this one in your hands. Take the first block of section 3, the one that asks the library to check `KillOrder`, and strip its promises: the block below is that one with no `#-->` line. Before you run it, write under each question the line you expect. Then run it and compare. Section 3 says why the last answer is what it is; whether it is the answer a function with that name should give, I leave to you.

HARO

```
cCode = "class Shop" + nl + "def KillOrder()" + nl + "  return" + nl
aFindings = StzCheckCode(cCode)
? aFindings[1][:rule]
? aFindings[1][:message]
? StzCodeIsClean(cCode)
```

✎ not run: an exercise, written for you to run

When you have settled it, you will know why a word rule advises and does not refuse.

The last chapter is about the next line. Its principle says when that line can be written: the record carries a solution only when it ran on its own engine and a named person said what they saw.

WHERE THE EVIDENCE LIVES

NARRATION	`stz-rules-as-graph-rules-narration.md`
NARRATION	`stz-governance-lineage-narration.md`
ON THE SITE	estate
ON THE SITE	principles
RULINGS OF THE STUDY	12.16, 12.20, 12.29

PART VII · CHAPTER 33 · SPINE

The next line: a solution on its own engine

DRAFT

Run on 2026-10-15 13:01 UTC+01:00 inside the library at commit 93d7a39bd: 3 of 3 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

In this chapter I show you the floor the next line will stand on. A platform is declared, built and deployed in that order; a twin touches nothing real; a last door refuses a fake service. Then I say what the three projects establish today, and I read from the record the five steps from that floor to the next line: a solution on Softanza's own engine, and what a named person who used it says. I say five things I refused, among them a reference without a document, my own figure for the bridge runtime, and an instrument as proof of a perception. The record carries a solution only when it ran on its own engine and a named person said what they saw.

1. THE CONCEPT, AS THE FIELD HOLDS IT

In our industry, a platform is proven when someone else runs it for real. The home page shows the logos of companies that use it; a case study tells one of them in two pages. A team that believes in its own product uses it for its own work, and a compiler that is serious compiles itself. Engineers who need a scale have one: NASA's readiness levels, later a standard, where the highest level is a system proven in the place it was built for.

These are reasonable tests, and they all point at the same thing: production. What they do not ask is who saw it work, and what that person said.

2. THE FIRST PRINCIPLE

The record carries a solution only when it ran on its own engine and a named person said what they saw.

3. THE SOFTANZA FORM

This is the last chapter, and it looks ahead. Let me first show you the floor the next line will stand on, because the floor exists and runs.

A solution is declared before it is built, and the platform that carries it refuses to be deployed before it is built. What does it answer when you ask it to deploy first? Run it:

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
}
```

✓ ran: printed what this block promises

The platform answers: `stzPlatform.Deploy: the platform must be Build() before it can be deployed.` Built, then deployed, `IsDeployed()` answers `1`.

Before anything is real, the solution can run on a twin, a copy of the system that holds no reference to reality. Where does a file written into the twin live? Run it:

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
```

✓ ran: printed what this block promises

It answers `virtual`, and `StzEngineFileExists` answers `0`: the file was written, read back, and is not on the disk at that path!

And at the production gate, a solution cannot reach production with a fake service bound in it, whoever asks. In chapter 1 you saw the registry refuse; here the delivery itself refuses, with an entitled person as the actor. What does the delivery answer when that person deploys to production with a sandbox bound? Run it:

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
}
```

```
✓ ran: printed what this block promises
```

The person was allowed to deploy. The delivery refused it! `WasCommitted()` answers `0`. Why did it refuse? The log's first error entry says only that the external surface is not production-ready; the second names the service: `sandbox-in-production @ shop/payments`. A payments service that is a sandbox cannot be what a real customer pays through. This is not a question of authority. It is a question of what is true.

So the floor is there: declare, build, rehearse, refuse the fake, commit through a governed actor.

Now what the projects show, and the step after them.

Has Softanza been used? Yes. I wrote that in chapter 1 and I hold to it. Parts of Softanza are in production, and others are in development. Three projects, and their feedback, shaped its design. Softibank, a bank in Niamey, has an organisation-assessment product I built beside the library, installed on its own network under licence. The installation guide delivered to the bank attests it. A restaurant in Lyon was delivered a first version of a platform for neighbourhood commerce. An amendment closed on 4 August; the notebook shows that the owner took part in the first demonstration, and on 19 August he made a day of field measurements with two phones. The last dated record of that project is 25 August 2026, and the next record is the one the next phase writes. An organisation in Niger has its hub designed. The site lists, project by project, each lesson it names and the file of the library that carries it. Part VI of this book tells their stories.

And the site says, in the same page, what those platforms run on today: ordinary stacks. The bank's is a Python application. The restaurant's is a set of web worlds, each a single file. The organisation's hub is designed, and its build is to come. The site says their declarations are being written as Softanza's own, so that moving them onto the Softanza engine is a change of engine and not of plan. The site's own words are exact: it is a step ahead, not a step taken. That step is the next one.

So the next line is this one: *the solution ran on Softanza's own engine, and this is what the user said.*

What leads from here to that line? Five steps, read from the record and not composed.

1. **A customer's declarations run on the engine instead of the ordinary stack.** The site says the declarations are being written as Softanza's own; I found no file of them yet, and chapter 31 says where I looked. The move to the engine is the next step.
2. **The language's own runtime runs the library on the customer's system.** Today the blocks of this book are run by the runtime of the former host language, version 1.27, and every page says so under its proof line. The bridge runtime that is meant to replace it was reported on 5 October 2026 to pass every assertion the old runtime passes on the library, all but one HTTP/2 request. That request failed once in the pass and passes twice when run alone. On every stable line, 3,461 of 5,112 programs agree with the old runtime. Those are two counts and not one: the first is of the suite's assertions, the second of whole programs, a program counted as agreeing only when every stable line it prints is the same on both runtimes. The morning's run gave 3,425, the figure chapter 27 prints; the 3,461 are the evening's, after a day of fixes. Its pass under Linux is the next step. Haro's charter was ratified on 1 October 2026, and Haro's own stages pass on its own fixtures. Whether Haro's own runtime runs the library is not yet measured.
3. **The solution writes its guarantee sheet.** The vision asks every solution for five rows, one for each system it answers to, technical, social, ecological, economic and cultural, judged as fixtures, with the machine derived from them. The sheet is designed, and the first one is yet to be written.
4. **Live hosts and a real board.** The deployment's verification against a real host or a real account is gated by infrastructure. The declared machine, a device written as a declaration, boots and judges itself in an emulator. Its hosted profile makes four promises: it is always reachable, it has a stable name, it keeps a durable log, and it survives a cut. The emulator can check three. The

fourth waits for a watchdog on a real board, a watchdog being, in the field's plain sense, a hardware timer that restarts a machine that stops answering. The site says no production workload is on it yet.

5. **A person perceives it, and the record says who.** Not a counter. A person, named, with a date and what they said.

The last step is a law of the estate, and it was earned. The sound plane of the library once held hundreds of passing assertions while, to my ear, it did not yet sound right. Since then, a piece of work that reaches a person through a sense must ship in a form a person can perceive, and the record must say who perceived it and what they said. And when nobody has, the law says it plainly: unperceived is a state, not a failure. A report may be honest and still be unperceived.

The record already holds the form of the line. In another plane, on 2 October 2026, the record says the Principal, its name for me (chapter 32 says so), listened to a sound the library made and said, in the words written down: "yes it's real drum!!" It is not a solution, and it is not a customer. But it is a person, a date, and a sentence, and it is what the next line will look like.

The record also holds two verdicts still to come. The pictures the library draws were judged by an AI model, forty-one pictures or text sections, twenty right, nine wrong, twelve uncertain. The record says that no human has looked at them yet; that look is the next verdict. The live run of the payments plane with the organisation's test account was postponed, by me, and reported as unperceived. Neither is a failure. Both are next.

THE DECISION, AND WHAT WAS REFUSED

4. THE DECISION, AND WHAT I REFUSED

I refused to be told the projects did not count. The first reading of the record wrote that no user outside my own projects had been measured. It was wrong, and I said so: Softanza has been used in real projects which, while not complete, gave the feedback that shaped its design. The reading was corrected in the record, in place, with the date. What replaced it is exact: the users exist, and their feedback shaped the design, and the next step is to move one of them onto the engine.

I refused a reference without a document. The site counts a project as a reference only when a document proves it, and it quotes no number of countries and no number of programmers. A lesson is listed only when a file of the library carries it.

I refused to move a platform by rewriting it. A customer's platform moves to the engine as a change of engine, not of plan. The declarations are to be written once, as Softanza's own, and the engine beneath them changes. I hold this as a view, not a result.

I refused my own figure. I have said aloud how far the bridge runtime has come, as a percentage. I do not print it without the guard that produces it. The stage word is "in construction", and the measured rows are the evidence.

I refused an instrument as proof of a perception. A phase whose evidence is entirely a counter is not done when the thing itself is something a person sees or hears. The counter is a result. The person is the verdict. The sound plane above, with its hundreds of passing assertions and the sound that was not yet right, is what stands behind it.

5. THE COURT

The floor of this chapter stands on three guards. `full_scope_model_narrated`, the architect's common ground, proves that a solution is modelled first, that a deployment before a build is refused and logged as an error, and that an AI model cannot perform the deployment. `virtual_system_twin_narrated` proves that rehearsal touches nothing real, that the plan narrates itself, and that committing is the one door. `service_delivery_narrated` proves that a production deployment with a fake bound does not run at all, with every reason named, and that the gate opens once the service is real.

The court of this chapter's subject is not a guard. It is a line in the record: who, when, and what they said. That line is the next one to be written.

Read from the library's test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. The figure is the number of assertions the guard holds, counted in its file; the site shows each one run.

`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. THE FIELD, AND WHERE IT LEADS

The field leads in production. Ordinary stacks run real workloads for real users today, a fact of the field and not a reading of the record; the record's own page names among them the bank's Python application, installed under licence. The

restaurant's first version is single-file web worlds, delivered, with a day of field measurements on 19 August, and the hub is designed.

The field leads in live infrastructure. Its tools drive real providers and real hosts every day; Softanza's deployment is proven end to end on local sites and rehearses the rest; live verification against a real host is the next step.

The field leads on hardware in the field. Its devices run in shops and clinics; the declared machine boots in an emulator, and its board is the next step.

The field leads on speed. The runtime Softanza measured itself against was faster on every row, by a factor between one and two, and the estate's note recorded the measure.

Where I think the record stands apart is in what it holds itself to. Every reference has a document. Every stage has a word. By the design document, the emulated deployment runs the same artefact that ships. A fake cannot ship and an AI model cannot commit. And a perception is a named, dated, quoted line, which a logo on a home page does not carry.

7. THE LIMIT

BUILT	a declared platform built, then deployed, and refused in the other order; the twin that touches nothing real; the production gate that refuses a fake service
DESIGNED	the runtime that is the language's own, running the library; the guarantee sheet of a solution in five rows; live hosts and a real board
ABSENT	a customer solution running on the Softanza engine, and a named person's verdict on it; the customers' declarations as files

This chapter looks ahead, so the limit is short.

What is built: the floor. Declare, build, deploy in order; rehearse on a twin; refuse a fake at the production gate; commit only through a governed actor.

What is designed: the runtime that is the language's own, running the library; the guarantee sheet of a solution in five rows; the verification on live hosts and on a real board.

What is absent: a customer solution running on the Softanza engine, a named person's verdict on it, and the customers' declarations as files (chapter 31 says where I looked). Beside it stand the floor above and the feedback of three projects. The record does not yet name the project that moves to the engine first, nor the person who will perceive it; I leave both unwritten until the record can carry them.

8. SIX DOORS

DOOR	WHAT YOU TAKE FROM THIS CHAPTER
The programmer	Your code runs today through build, rehearsal and a governed deployment. What comes next is a customer running it on the engine, and the record will carry that claim the day it is true.
The analyst	Ask for two things before you count a project as proof: a document, and a named verdict.
The designer	What you design is judged last by a person's sense. An AI model's opinion of a picture is a result; a person's is the verdict.
The decision-maker	Read the stage word before the promise. Today the floor is built; the customer run on the engine is the next milestone, and its stage word is absent.
The educator	Teach that unperceived is a state and not a failure. An honest report can ship on it, if it says so.
The agent	You may rehearse a deployment and explain it. You will be refused when you try to perform it (chapter 24 shows the refusal).

9. EXERCISE

Declare a platform for a project you own, with one server. Before you run anything, write the line you expect under each of three questions: what happens when you ask it to deploy before you build it; what happens when an assistant asks to deploy after the build; what happens when a person does.

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
```

✎ not run: an exercise, written for you to run

Run it. Then this one, which I also left in your hands. Here is the production-gate block of section 3 with one line changed: the actor is no longer the person but the assistant, `LLMActor("assistant")`, as in chapter 24. I did not run this version, and I print no answer for it.

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]
```

✎ not run: an exercise, written for you to run

Run that one too. Read what `WasCommitted()` answers, and read the log: the block of section 3 printed its second error entry for the person, and I make no claim about the assistant's. Section 5 says what the three guards prove, and section 8 what an AI agent is refused. When the gate answers, you will know what it refuses for what is true and what it refuses for who is asking.

Every block in this book is yours to run again. Now write one more line, the next one for your own project, in the record's form: who perceived it, on what date, and what they said. If nobody has, write "unperceived", and say why. That line is the

end of this book, and the beginning of the next one.

WHERE THE EVIDENCE LIVES

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`
ON THE SITE	customers
ON THE SITE	forged
ON THE SITE	platforms
ON THE SITE	estate
RULINGS OF THE STUDY	12.5, 12.10, 12.13, 12.21, 12.29, 12.30

Appendices

APPENDIX A

The register of innovations

DRAFT

This page has no code to run.

My study of the record read four inventories the library keeps of itself: the narrations with their titles, the design documents, the engine's phase of paradigm engines, and an outside reader's list of rethought concepts. It crossed them, and found forty-six innovations in seven families. This is that register, as the study ruled it on 7 October 2026, with one column of this book's own: the chapter that tells each row. A later ruling the same day adds a forty-seventh row, answer by example, with the stage words of a plan: nothing induces yet, as chapter 18 says. It is not in this table, which keeps the forty-six the study counted.

Read the register as the study reads it, and no better. It is a reading and not a census. A concept with no narration and no design document of its own was not counted. The study's evidence counts files that name the thing in code outside tests and archives, test files, and the engine module where one exists; it counts files that name a thing, and not tests that pass. I have not re-counted any of them. The study also counted how many pages of the site named each innovation; the site has changed since, and I leave that column out.

Of the forty-six, eight are not told in this book: the adverb engine, named variables, the unified find, object history, the intent-based file interface, natural folder navigation, the invisible and the lookalike, and statistics with the linear solver and matrices. They are in the register because the record holds them, and untold here because this book had to choose.

A · PARADIGMS

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE (7 OCTOBER 2026)
1	Scope-Oriented Programming	behaviour with an invisible governing frame, cured in four moves; four instances	8, 23	a design of 450 lines; instances in regex, system, delivery
2	The Softanzuter ladder	pattern, Xuter, Softanzuter, AI agent	9, 17	two engines, fifteen blocks, an agent mind
3	Reaxis	reactive programming redesigned as Container, Stream, Rfunction	9	ten code files, fifty-nine test files, two engine modules
4	Near-natural programming, NNL and `stzNatural`	close enough to natural language to be readable, structured enough to execute	13	three narrations, two design documents, a folder of thirteen files
5	Knowledge programming, `.zknw`	the semantics of thought: one knowledge file, an operational system	14	a narration that runs; the knowledge graph in seventy-one files
6	Functions as linguistic expressions	twenty-two forms that encode computational semantics	8	about six thousand tags
7	The four metaphors	walker, checker, yielder, performer	6	three narrations
8	Conditional code, W and WF	a condition travels inside an argument	6, 12	a narration
9	Future actions	plan a sequence first, then execute it all at once with natural timing	8 (removed on purpose, 10 July 2026)	a narration of seventy-five lines; two test files
10	Refinement-centred programming	refinement points, a four-stage gate, a file of refinements	10 (four stages in the design, five checks in the code)	two deliveries of the refinement gate, with their guards
11	Wise coding	Softanza asks; expression is free, admission is governed	18	two guards
12	Programmatic intelligence, agents that cannot hurt you	the mind programmatic by default	17, 20	eight suites

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE (7 OCTOBER 2026)
13	The domain language model, constructive prediction	the next thing a model says is a checkable unit of the world	19	a guard; a contract

B · THE TWELVE PARADIGM ENGINES

The study counted twelve engines; the register gives nine of them a row of their own, and the end of this page returns to the twelve.

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
14	The truth engine	truth is not binary: domain-configurable truth, semantic falsification	named in 3; its neighbour, the truth chain, is in 13	three code files, three tests; the engine marked done
15	Probabilistic quantifiers	few, some, most, all, with configurable thresholds	3 (some, most and few draw at random, and I say so)	five code files, thirteen tests; the engine marked done
16	The adverb engine	from quick to quickly: morphology for `Explain()`	not told	one code file, no test
17	The timeline	a temporal workspace: points, spans, blocked regions, constraint reasoning	9, 15	seven code files, sixty-nine tests
18	The grid navigator	a two-dimensional cursor with direction and boundary awareness	9	ten code files, forty tests
19	Section merging	interval arithmetic, three merging strategies	9 (the verb the article presents is absent)	five code files, no test
20	Deep operations	path-based recursive traversal of nested lists	9	fourteen code files, thirty-seven tests
21	Named variables	dynamic names; named objects	not told	one code file, no test
22	The polyglot bridge, EXCIS and ExterLib	orchestrating a polyglot symphony	16	four narrations, a design of 1,986 lines

C · PATTERNS AND REACTION

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
23	The six pattern languages	regex elevated to lists, numbers, tables, matrices, time and graphs	15	13,385 lines; eight narrations
24	RegexMaker and RegexData	regex declared, not written	15	two narrations
25	The unified find	one find across media	not told	a narration

D · EXPRESSION AND PERCEPTION

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
26	Visual orientation: `viz`, boxes, string art, the plotting system	where ASCII art meets dataviz	7 (the viz prefix only)	seventy-seven files for viz
27	The invisible and the lookalike	unmasking hidden Unicode characters and lookalikes	not told	two narrations
28	Object history	the chain's trace	not told	a narration that runs
29	The string and list idiom	lists in strings, the small operators, duplicates, smart builders	7	ten narrations
30	The intent-based file interface	beyond file modes	not told	a narration of 552 lines
31	Natural folder navigation	state follows action	not told	three narrations

E · DATA, TIME AND REASONING

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
32	Time programming	time, date, duration, calendar capacity planning	9, 15 (the timeline and Timex only)	six narrations

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
33	`stzGraph` and its organs	twenty-five notations, a planner, a query, an organisation chart	14, 29, 30	eight narrations
34	Rules as graph rules	rules that see the whole picture	29, 30	a design plan; a narration
35	Numbers rethought	a global rethink of numbers, exact by default	22	a design of 3,167 lines
36	Statistics as a language of thought, the linear solver, matrices	the study gives no line for this row	not told	four narrations

F · SYSTEM, DELIVERY AND OPERATIONS

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
37	The System Foundation and the virtual twin	three scopes over one solution; rehearse, then commit	23, 24	guards; the engine
38	Emulate, deploy, provision; dev to deploy	the whole solution running before it ships	24	four narrations that run
39	Service virtualization	code first, subscribe later; production refuses a fake	24, 28	a narration; the registry
40	The performance plane	the honest stopwatch, the frame profiler, the governed loop, judgment, the tail	21 (the frame profiler is not shown)	ten narrations
41	The security plane	the event, the ledger, the sentinel, the drill, the response, attestation	21	eight narrations; an incident analysis of 815 lines
42	Governance lineage	answerable forever	30	a narration
43	The front door and secrets	authentication and credentials, self-redacting	21 (secrets; the front door as an event only)	two narrations

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
44	Structured logging, the engine senses, the XML parser that says no, safe hashlist access	the study gives no line for this row	27 (the engine senses)	four narrations that run

G · THE SELF-KNOWING LIBRARY

#	INNOVATION	THE RECORD'S ONE LINE	IN THIS BOOK	THE STUDY'S EVIDENCE
45	The information-tagging strategy and self-documentation	every unit retrievable, explainable, conversational	7 (the tags, and that no court reads them)	a design of 238 lines
46	Locale management and culture	software of global reach, the Softanza way	22 (the loss: no locale layer)	a narration that runs

WHAT THE REGISTER SAYS

The study's reading, which I keep: what the site names is what has a product page, a principle or an area of the Atlas. What it leaves in the dark is what cuts across the areas. Every paradigm of the first family except rows 4 to 8 and 10 to 12, which the site already named; all twelve engines except the polyglot bridge; the pattern languages as a family; the idiom. This book is in part the answer to that: its parts are cut across the areas, and not along them.

The study also reconciled the twelve engines. The macroplan marks the phase done. Eight of the twelve have a source file under their name in the engine; four have none, which are truth, quantifiers, polyglot and polycode, the first two reported done as libraries. On the library's side three engines have no test file at all: adverb, section merging and named variables. Two of the three are rows 16 and 21 above, which this book does not tell, and the third, row 19, it tells as an absence.

WHAT IS NOT MEASURED

No code was run to make the register. The evidence column counts files that name a thing, and the engine's done marks are the macroplan's. Whether an innovation's narration runs once its stale names are repaired was not tested. A row's chapter in

the fourth column is the chapter that tells it, and a row that says "named in" is a row the chapter mentions and does not tell.

> WHERE THE EVIDENCE LIVES
>
> RULINGS OF THE STUDY 12.19

APPENDIX B

The fourteen house rules

DRAFT

Run on 2026-10-15 13:01 UTC+01:00 inside the library at commit 93d7a39bd: 2 of 2 blocks kept their promise.

The blocks were run by the runtime of the former host language, version 1.27; Haro does not run the library yet. The engine binaries were built in a checkout of this commit, newest built 2026-10-05 20:27 UTC+01:00; 1 engine change(s) committed after that build are not in them.

The library has a court over its own code. It reads a class as a graph (five rules) or as text (nine) and judges it against rules, some of them written after a defect of that shape was found in the library's own code. This appendix lists the fourteen, and shows three of them firing.

There are two fourteens in this book, and they are different sets. The fourteen **house rules** of this appendix are rules of the library's code: they are a court, and they run. The fourteen **rules of writing**, in chapter 7, are a reading of how the style is written. They are readings of the code, stated as rules, and no machine judges them. The coincidence in the number is a coincidence.

The list of names is fourteen and the library can print it. One more text pass, `for_in_loop`, also reports, and so do two rules over a whole project, `no-dead-code` and, in the deep audit, `no-cyclic-calls`:

HARO

```
? len( StzCodeRuleNames() )
#--> 14
? StzCodeRuleNames()[2]
#--> q-returns-object
? StzCodeRuleNames()[14]
#--> value-called-as-function
```

✓ ran: printed what this block promises

Here are three of them, fired on code I planted. A method that writes to the console is a warning, so the code stays clean. A method that differs from another only by case is an error, since the host language does not tell the two apart and the file would not load. And a class that defines `Len()` shadows the host's own function, and is refused:

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
```

✓ ran: printed what this block promises

A note on spelling, which a program that reads findings must know. The list uses hyphens: `library-prints`. A finding of eight of the fourteen comes back with underscores: `library_prints`. The findings of the five rules over the code graph (rows 1, 3, 4, 5 and 6) keep their hyphens, and so does the finding of row 2. I print what the run printed.

THE FOURTEEN

#	RULE	WHAT IT REFUSES	SEVERITY
1	`no-len-method`	defining `Len()` on a class: it shadows the host's builtin and breaks every caller	error

#	RULE	WHAT IT REFUSES	SEVERITY
2	`q-returns-object`	a method named with a closing `Q` that does not return an object to chain	error
3	`no-aggressive-verbs`	method names that begin with Kill or Destroy: the Softanza tone	warning
4	`engine-first`	new code that reaches for the host's byte-oriented string functions where the engine's forms exist	warning
5	`q-has-plain-twin`	a chaining mutator with no plain twin: `Q` chains, and plain acts	warning
6	`no-case-collision`	two methods of one class that differ only in case: the host is case-insensitive, so the file fails to load	error
7	`dead-knob`	an attribute that a setter writes and nothing reads: a setting that can change nothing	error; warning when only its own getter reads it
8	`setter-resets-on-reject`	a setter that answers a value it dislikes by overwriting what was already there	warning
9	`setter-only-moves-one-way`	`max()` or `min()` against the very attribute being set: a ratchet that swallows the other direction	error
10	`misspelled-name`	a method with a known misspelling while the correct name does not exist	warning
11	`library-prints`	a class method that writes to the console while it works; methods whose purpose is to display are exempt	warning
12	`empty-method-body`	a method declared and doing nothing; harsher for a predicate, which then answers nothing, read as false	warning; error for a predicate
13	`writes-a-mutable-constant`	assigning to `true`, `false`, `null`, `nl`, `tab` or `cr`: in the host they are ordinary globals	error
14	`value-called-as-function`	calling the result of a call, the shape a global replacement of tokens leaves	error

I fired rows 11, 6 and 1 above, in that order. Chapter 32 fires row 3, chapter 7 fires rows 10 and 13, and chapter 5 fires row 2. For rows 7, 9 and 12 I read the severities from the source, as I did for rows 4, 5, 8 and 14; I did not fire these seven.

WHAT THE COURT DOES NOT JUDGE

None of the fourteen is a rule of writing. No machine judges, today, the style of a program written with the library. That is chapter 7's limit, and it is the reason the other fourteen are in this book as a table with their state, and not as a court.

THE OTHER FOURTEEN: THE RULES OF WRITING, AND WHERE EACH STANDS

Read from the study's ruling on the style, with the state of each at the commit this book runs.

#	RULE OF WRITING	STATE
1	A value is lifted to its object by `Q()` or a typed lifter	runs
2	A scene is a brace block on one object	runs
3	Every printed line carries its answer under `#-->`	the promise is judged when written; a line without one is not refused
4	A parameter is named, and the name reads as English	runs, and in any order
5	A chain mirrors the order of thought, one step per line	runs
6	The verb's form says what happens	runs
7	Suffixes and prefixes are small and mean one thing each	in part
8	A function has alternatives for the way a person says it	in part
9	Precision over generality, when it costs nothing	in part
10	Code travels inside an argument as a small string	the seat runs; the mark on a parameter's name is retired
11	Small literal languages live in strings	runs in one form

#	RULE OF WRITING	STATE
12	A sentence can be near-natural	not as the deck wrote it
13	A function may be named in any human language	Chinese and French names raise; the Arabic one was not run
14	A comment speaks before the code, in the teacher's voice	runs

WHERE THE EVIDENCE LIVES

RULINGS OF THE STUDY 12.16

APPENDIX C

The twenty-two verbs and the constitution

DRAFT

This page has no code to run.

The interface constitution lives in its own repository, and the site says that repository is not public yet. This book calls it the interface constitution and does not name the product it belongs to. The library holds only the slot for its verbs and the finder of the elements that serve none; the list itself is the constitution's. So this appendix prints what I read from the constitution's own file on 9 October 2026: its version is 3.11, last updated on 16 August 2026, with seven articles, thirty-one sets and one hundred and twenty-two rules. The counts in this sentence are the file's own, and I counted none of them myself.

Chapter 26 shows how the library finds an element that serves no verb. Chapter 25 shows what a new medium adds. Chapter 20 shows the right to a human as the end of the loop.

THE FOUR AXIOMS

Its four axioms are calm over power, intent over mechanics, visible over hidden, business over technology. Its preamble says business software is not entertainment, and that its interface should not impress, and should serve.

THE SEVEN ARTICLES

ARTICLE	TITLE	WHAT IT HOLDS
I	Calm	the interface must never compete with the operator's thinking; calm is a functional requirement and not an aesthetic choice

ARTICLE	TITLE	WHAT IT HOLDS
II	Intent Over Mechanics	interaction starts from what the human is trying to do, and widgets are consequences of intent, never primitives; selection never implies execution
III	Visible State	if the system knows something important, the user must see it: active filters, pending drafts, norms, background work, machine provenance
IV	Reversibility	every act is reversible, or explicitly and visibly irreversible: staged, named and reviewed
V	Predictable Authority	every consequential act names its actor, and an intelligent agent is an actor under this law, never an authority above it
VI	Operator Primacy and Vibes	business users are operators, not explorers; vibes, densities, dialects and surfaces are equivalent vibes of one truth
VII	Amendment	the law grows only from evidence; removing a protection requires proof the harm is gone

THE TWENTY-TWO VERBS

The grammar is closed: rules quantify over these verbs, and a new medium adds renderings, never verbs. Six families, each with its invariant, as the constitution writes them.

FAMILY	VERBS	INVARIANT
Orienting	discover, understand, locate	discovery must be satisfiable at every entry point
Attention	focus, filter, compare	an active filter is visible state; focus and discovery cannot both be at maximum
Information	read, scan, scroll, zoom	a read surface that silently acts is an intent violation
Selection	select, highlight, preview, mark	selection expresses interest, never commitment
Action	act, confirm, cancel, undo	every act has a reachable undo, or declares why it cannot
Continuity	pause, resume, retry, exit	pause, resume and exit are always reachable; a trapped flow is unconstitutional

That makes three, three, four, four, four and four: twenty-two. Of them, the library's worlds (chapter 26) know five as screen intents: discover, understand, focus, select and act. The library does not check the word.

THE SIX RIGHTS

RIGHT	WHAT IT SAYS
to understand	every element, denial and setting can explain itself in the operator's language
to reverse	what was done can be undone, and what cannot says so before, not after
to know	the state of the system, filters, drafts, progress and provenance, is always visible
to leave	every flow, focus and overlay has a reachable exit that returns the user to where they stood
to act without fear	no element creates anxiety through unpredictability, irreversibility by default or hidden consequences
to a human	wherever a machine serves, a person is reachable, clearly and without penalty

The last right is enforced by a single rule, and that rule is marked as a matter of judgment: a person decides whether it is met, and no machine checks it. The library does not mention it.

INTELLIGENCE UNDER LAW

Set 31 is the constitution's answer to the AI agent, and it is where chapter 20 meets it. Its ten rules, with how each is enforced, as the file records it:

RULE	TITLE	ENFORCEMENT
97	The Agent Is an Actor	judgment
98	Proposals, Not Commits	judgment
99	Visible Provenance	a machine, later
100	Deterministic Islands	a machine, later
101	The Dignity Contract	judgment
102	The Right to a Human	judgment
103	Grounded Proactivity	judgment
104	Machine Legibility	a machine, later

RULE	TITLE	ENFORCEMENT
121	The Assistant Never Masks the Work	judgment
124	The Generator Ships a Working Thing	judgment

Rule 98 is the law of chapter 20's plan, written for an interface: a proposal is not a commit. Rule 99 asks for the mark on machine-made content; chapter 14's provenance is the nearest thing the library builds. Rule 104 asks for machine legibility; chapter 26's accessibility tree is the nearest thing the library builds: what a screen reader can operate, an AI agent can operate. Seven of the ten are enforced by judgment, and three by a machine that is to come.

WHAT THE CONSTITUTION FORBIDS

The constitution lists seventeen forbidden patterns, each tied to the rules it violates. Among them are modal storms, invisible active filters, hidden destructive actions, and selection that silently executes. Its retired numbers are never reused, and two numbers are burned and reserved for ever.

WHAT THIS BOOK TAKES FROM IT, AND WHAT IT DOES NOT

It takes the closed grammar, which it uses in chapters 25 and 26; the six rights, of which it uses one; and the sets, of which it uses the one for AI agents. It takes also the rule on the five values of status, which chapter 25 renders as colour, earcon and phrase. It does not take the rest. No guard of the library judges a shell against the verbs, and the library's own design text still lists twenty-four verbs from an interaction engine that I removed on 26 May 2026. The constitution's list is twenty-two.

> **WHERE THE EVIDENCE LIVES**
>
> RULINGS OF THE STUDY 12.23, 12.24, 12.25

APPENDIX D

The glossary: words name states, never people

DRAFT

This page has no code to run.

A word in this book names a state of a piece of work, or a role held in a scope. It does not name a rank, and it does not name a person. A crafter is someone who is declaring something, and a person is a crafter on the day they declare. An architect is what one holds in the scope of a solution. A user is whoever is using. When a sentence of this book is about a person, it says what the person did, on what date, and in whose words.

Each entry gives the word as the book uses it, and the chapter where it is shown. Where the record uses a word in two senses, both are given, and they are numbered.

absent. One of the three stage words. What the record does not hold, in code or in files, as far as I looked, and I say how far. Chapter 3.

admission. The act by which a candidate becomes knowledge: it is judged by the world's rules, and refused with a reason if it fails. Expression is free, and admission is governed. Chapters 17, 18.

agent. A program that proposes and cannot commit. In the library an AI agent is a file, judged at load, whose AI model, if it has one, holds no capability to produce effects. It names an actor and never a person. Chapters 17, 20.

architect. What one holds in the scope of a solution: the profile, the worlds and the governance. Not a rank. Chapter 23.

authority. The level at which an actor may act, compared with the risk of the act. Distinct from permission: permission is held, and authority must cover the risk. Chapter 30.

bond. A declared relation between two worlds of a constellation, for one named act. Nothing crosses without one. Chapter 23.

built. One of the three stage words. The code exists, a guard exists, and a block ran. Chapter 3.

candidate. What a reply, an AI model's answer or an example is, until the court has judged it. Chapter 18.

checkpoint. A refusal kept for a person, with its context and a lifetime. Chapter 18.

commit. The one act that reaches reality. Only a governed actor commits, under a scope, and the AI model cannot. Chapters 20, 24.

commons. The shared ground of a constellation, what every world of it stands on: an identity, a session, a store. It is a world like the others, world zero, and not a special kind of node. Chapters 4, 23.

constellation. A graph of worlds, with bonds between them and governance over every crossing. Chapter 23.

court. The judge of a thing. A court says kept or refused, and says why in the words of the declaration. A language is born with its court. Chapters 5, 11, 12, 21.

crafter. Whoever declares what they know about a world, and owns what they declare. Skill is not part of it. Chapter 2.

declare. To say what a thing is before it is built, in a text a court can judge. The act of the kernel. Chapters 3, 12.

designed. One of the three stage words. A document says so, and nothing runs. Chapter 3.

door. In this book, two senses. (1) The *doors* of a chapter: the programmer, the analyst, the designer, the decision-maker, the educator and the agent, six ways in which a chapter is useful, each a way of using the work and not a job title. (2) The *door* of the foreign runtimes: one governed way to run code in a language the platform does not speak. A door is a runtime and never a syntax. Chapters 2, 16.

engine. The Zig library that does the work once, beneath every surface. Chapter 27.

estate. My body of projects: some twenty repositories that depend on each other, the library among them, written on one machine under one protocol. Chapter 32.

facet. In this book, three senses, which chapters 12, 16 and 26 keep apart. (1) A surface of a declared language: its text, its court, its mouth, its voice, its tongues. (2) A syntax of Haro that a programmer of another language could write; the record calls this a face; this book says facet, and chapter 12 defines it. (3) A tool of an environment's ribbon, ordered by the life of the thing being made. A facet is a syntax and never a runtime.

fake. A double that says it is one: a sandbox, a mail sink, a recorded answer. A fake must never ship. A double that says nothing about itself is taken for live, and passes the gate. Chapter 24.

floor. One level of the platform's picture, from the engine at the bottom to the organisation at the top, each with its judge beside it. Not a layer of the library (core, base, max) and not a plane. Chapter 4.

forge. The machinery that produces a language's court, mouth, printer and grammar from its declaration alone. Built outside the library, designed for it. Chapters 11, 12.

gap. What a goal needs and the world does not yet hold. A conversation that asks starts from the gap. Chapter 17.

gate. A place where something is refused before it goes on. The registration gate, the production gate, the capability seam. Chapters 20, 24.

goal. A wanted state of a graph. The gap is measured, and a goal that cannot be evaluated is refused. Chapters 17, 23.

governance. Permission, authority and risk, compared before an act, fail-closed, with the decision kept. Chapters 20, 30.

guard. A file of the library's test tree that holds assertions, and that proves a chapter. This book counts a guard's assertions and does not run most of them. Chapters 5, 21.

Haro. The language this book writes in: the surface of the former host language, kept and renamed, with a charter and a runtime still in construction. Haro does not run the library yet. Chapters 3, 27.

house rules. The fourteen rules the library runs over its own code, five over a graph of the code and nine over its text: a court an AI agent can run before it commits. Not the fourteen rules of writing of chapter 7, which no machine judges. Chapters 5, 32; Appendix B.

ledger. The security plane's record of what was refused and why, written by the court and never by the thing it judged. Chapters 20, 21.

lexicon. The dictionary of words a sentence may use, each mapped to a verb the library documents. Chapter 13.

lineage. The record of a decision with the authority and the risk of its moment. Chapter 30.

LLM. Large language model: the AI model the field builds today, trained on text and asked to propose. In this book it is a proposer, never the mind. Chapters 1, 17.

narration. The author's article under the library's documentation folder, whose code is the old surface. Not an executable test. (The word has since been given to the executable guards named *narrated*, and chapter 21 says why that confuses.) Chapters 3, 21.

near-natural chain. A sentence in which every link is a real call, so that the human is the parser. Chapter 13.

pack. A human language as a block of data. Chapter 13.

perception. What a person has seen or heard of a thing, named, dated, and quoted. A comparison proves sameness and never rightness. Chapters 21, 33.

plan. The one artefact that crosses from a twin toward reality. It narrates, flags and validates, and is committed by a governed actor. Chapters 20, 24.

plane. One area of the library, named by what it governs and proved by its own guards: the system plane, the agentic plane, the planes of security and performance. Not a floor, and not a plan, which is the artefact a machine proposes. Chapters 1, 21.

platform. The envelope around a solution, and the ground a constellation stands on: capabilities, a commons, a registry, shells, a profile. Chapter 23.

posture. What the registry says a service is: sandbox, local, live, conformance. And for foreign code, what its executor says it is: trusted, external or sandboxed. Chapters 16, 24.

Principal. The record's name for me, the one who rules: a ruling is written only in my presence, and a session that rules on my behalf marks it so. A role, not a rank. Chapters 32, 33.

promise. The line `#-->` under an expression, which says what it prints. A claim about a stop of the program, judged kept, diverged, unreached or unjudged. Chapters 1, 21.

proposal. What an AI model, a reaction or an AI agent offers. It commits nothing. Chapters 17, 20, 26.

record. The repositories, sessions and files this book ran against, at the commit printed under every proof line. What the book reads its facts from, and never composes. The preface; chapter 32.

refinement. The write surface: a change declared with its points and judged by a gate of five checks. Chapter 10.

register. In this book, two senses. (1) One of the five forms a reply can take: an option, a data structure, a formula, a sentence, examples. (2) The register of innovations, Appendix A. Chapter 18.

rehearse. To run on what touches nothing, and to read what it would have done. Chapter 24.

reversibility. The class of an act: reversible, compensable or irreversible. The harder an act is to undo, the more trusted the code performing it must be. Chapter 20.

scope. A named frame in which a system act is written: development, deployment, runtime-current. The system acts of a program are always in a scope. Chapters 8, 23.

session. One run of an AI model at work on the estate: it proposes, builds, measures and reports in a fixed form of eight keys, and closes with a cost line. I rule; it does not. Chapter 32.

spine. The nine chapters that carry the whole argument and can be read alone. The preface.

stage word. One of built, designed, absent, printed at the head of each chapter's limit. Chapter 3.

stamp. The time a session writes on a message, a memo or a commit. Read from the clock, never composed; an instrument compares every stamp with its commit and publishes the rows. Chapter 32.

study. My study of the record: a reading of the repositories, sessions and files, from which this book takes the register of Appendix A and the figures it marks as the study's. A reading, not a census: where the book re-ran a thing, the run is the evidence. The preface; chapter 3.

twin. A copy of a part of reality that holds no reference to reality: a twin of files, a twin of the process environment. Chapters 20, 24.

unperceived. The state of a thing that reaches a person through a sense, when no person has perceived it and said so. It is a state and not a failure. Chapters 21, 33.

wise coding. The platform asks the question the person has to answer, and refuses to guess the answer: Softanza asks and the world judges. The inversion of the loop of prompting an AI model and keeping what runs. Chapter 18.

workbench. Where a proposal is rehearsed. It holds a twin, a copy of the system with no reference to the real one; what the proposal would write goes into the twin, and what comes out is a plan. The one a programmer already uses, not a second one for machines. Chapters 1, 20.

world. One application, as a declaration with a being, a becoming and a way of being met. Chapter 23.

APPENDIX E

The guards and fixtures cited, by chapter

DRAFT

This page has no code to run.

A guard is a file of the library's test tree that holds assertions. A chapter that says "this is proved" names the guards that prove it, in the front matter of its source, and the build reads each one: it finds the file in the library at the commit the book runs at, and counts the assertions in it. A guard that is not there fails the build.

This appendix is that reading, by chapter. No figure in it is typed. It is regenerated at every build, so the figures are always the ones the last build read.

Two things to keep in mind. A figure is the number of assertion calls the file holds, counted in its text; it is not a run. Most of these guards are not run by the book, because they write files, open ports or spawn processes. The few I ran myself are named in the chapters that rely on them, with the date. And the only fixtures the book cites are the ones the guards and the blocks read: the course folder of the library's education program, and the data folders its guards name. The blocks of the book are themselves the other fixtures: each one runs inside the library at the commit named under every chapter's title.

The chapters cite 114 distinct guards, holding 3359 assertions between them, read from the test tree at commit 93d7a39bd on 2026-10-15 13:01 UTC+01:00. A guard cited by two chapters is counted once in this total. Names and paths are printed as the library spells them; the extension is the former runtime's until Haro runs the library.

1 · CHANGE ARRIVES FASTER THAN TRUST

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE SOFTWARE CRAFTER

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE METHOD, FROM FIRST PRINCIPLES

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE PLATFORM IN ONE PICTURE

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
00_stzapp_narrated	3	base/test/app/00_stzapp_narrated.ring
06_stzsuperapp_narrated	21	base/test/app/06_stzsuperapp_narrated.ring

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · LAW: THE CODE ADHERES TO THE WAY A PERSON THINKS, AND THE COURT JUDGES IT WHOEVER WROTE IT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE MENTAL MODEL, AND THE METAPHORS

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE WRITING STYLE, AND THE SEVEN GOALS

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
coderule_narrated	77	base/test/meta/coderule_narrated.ring

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · FUNCTIONS AS LINGUISTIC EXPRESSIONS, AND SCOPE

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · PATTERNS OF THOUGHT: THE SOFTANZUTER, REAXIS AND THE ENGINES

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · PROGRAMMING BECOMES REFINEMENT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · LAW: A LANGUAGE IS DECLARED AND CLOSED, AND WHATEVER SPEAKS IT, PERSON OR MODEL, IS JUDGED BY IT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
constrained_decoding_narrated	50	base/test/neural/constrained_decoding_narrated.ring

12 · A LANGUAGE, DECLARED

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · NATURAL, AND EXECUTABLE: TWO DIRECTIONS, ONE DISCIPLINE

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · KNOWLEDGE PROGRAMMING

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · PATTERN LANGUAGES

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · SEVEN LANGUAGES, ONE DOOR

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · LAW: PROGRAMMATIC BY DEFAULT, THE MODEL LAST, ADMISSION OVER GENERATION

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
dlm_narrated	16	base/test/learning/dlm_narrated.ring

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
wisecoding_narrated	13	base/test/conversation/wisecoding_narrated.ring
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 · THE FIVE REGISTERS, AND CONVERSATIONS THAT ASK

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE DOMAIN LANGUAGE MODEL

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · AGENTS THAT CANNOT HURT YOU

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
safeworld_narrated	76	base/test/agentic/safeworld_narrated.ring
agentfile_narrated	88	base/test/agentic/agentfile_narrated.ring
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 · THE COURT: PROMISES, PERCEPTION, SECURITY AND PERFORMANCE

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · COMPARED, HONESTLY

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
numeric_exactness_narrated	39	base/test/number/numeric_exactness_narrated.ring
numeric_no_loss_narrated	31	base/test/number/numeric_no_loss_narrated.ring

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · A WORLD, A CONSTELLATION, A PLATFORM, AND THE ARCHITECT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · REHEARSE, THEN COMMIT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · YOUR USERS KEEP THEIR MEDIUM

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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
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 · THE ENVIRONMENT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE ENGINE BENEATH: THREE LAYERS, ONE ENGINE, HARO

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
threat_model_narrated	11	base/test/security/threat_model_narrated.ring

28 · A RESTAURANT IN LYON

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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
judgment_narrated	41	base/test/perf/judgment_narrated.ring
graphrule_object_narrated	34	base/test/graph/graphrule_object_narrated.ring

29 · A BANK IN NIAMEY

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · AN ORGANISATION IN THE FIELD

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
governance_lineage_narrated	37	base/test/governance/governance_lineage_narrated.ring
01_platform_narrated	31	base/test/platform/01_platform_narrated.ring

31 · WHAT THE PROJECTS SHOW, AND WHAT COMES NEXT

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
service_registry_narrated	93	base/test/system/service_registry_narrated.ring
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 · MANY MINDS ON ONE BODY OF WORK

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
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 · THE NEXT LINE: A SOLUTION ON ITS OWN ENGINE

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
full_scope_model_narrated	62	base/test/system/full_scope_model_narrated.ring

GUARD	ASSERTIONS	WHERE IN THE LIBRARY'S TEST TREE
virtual_system_twin_narrated	48	base/test/system/virtual_system_twin_narrated.ring
service_delivery_narrated	60	base/test/system/service_delivery_narrated.ring

APPENDIX F

The narrations the book argues from

DRAFT

This page has no code to run.

A narration is one of the author's articles, under the library's documentation folder: an essay that tells an idea with its code. The table below counts them at the pin, and the book argues from the ones listed here. The code in them is the old surface, which Haro keeps under a new name. A narration is not an executable test: the word has since been given to the executable guards whose names end in *narrated*, and chapter 21 says why that is confusing.

Each chapter names its narrations in its source, and the build checks that each is a file of the library at the commit the book runs at. Where a name is not a file of that folder, this appendix says so. A narration's blocks are not the chapter's blocks. Several no longer run at that commit, and each chapter that cites one says which part, and runs its own.

The library's narration folder holds 135 at the pin. The chapters cite 50 distinct narrations. Every name is a file of the library's narration folder at commit 93d7a39bd. Names and paths are printed as the library spells them; the extension is the former runtime's until Haro runs the library.

2 · THE SOFTWARE CRAFTER

NARRATION	IN THE LIBRARY AT THE PIN
`stz-built-on-real-world-need.md`	yes
`stz-service-virtualization-code-first-subscribe-later.md`	yes
`stz-agents-that-cannot-hurt-you-narration.md`	yes

NARRATION	IN THE LIBRARY AT THE PIN
stz-rules-as-graph-rules-narration.md	yes
stzOrgChartDoc-BankUseCase.md	yes

3 · THE METHOD, FROM FIRST PRINCIPLES

NARRATION	IN THE LIBRARY AT THE PIN
stzstring-emptiness-narration.md	yes
istrue-softanza-truth.md	yes
stz-built-on-real-world-need.md	yes

4 · THE PLATFORM IN ONE PICTURE

NARRATION	IN THE LIBRARY AT THE PIN
stzApp-building-your-first-app-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes
stz-emulating-the-whole-solution-narration.md	yes
stz-vs-qt-unicode-narration.md	yes

5 · LAW: THE CODE ADHERES TO THE WAY A PERSON THINKS, AND THE COURT JUDGES IT WHOEVER WROTE IT

NARRATION	IN THE LIBRARY AT THE PIN
stz-mental-mode-narration.md	yes
stz-bridging-minds-and-code.md	yes
stz-functions-as-linguistic-expressions.md	yes
stzobject-history-narration.md	yes
stz-near-natural-language-programming.md	yes

6 · THE MENTAL MODEL, AND THE METAPHORS

NARRATION	IN THE LIBRARY AT THE PIN
stz-mental-mode-narration.md	yes
stzwalker-beyond-loops-the -walker-metaphor.md	yes

NARRATION	IN THE LIBRARY AT THE PIN
stzwalker-walking-at-the-edge-narration.md	yes
stz-conditional-code-narration.md	yes
stz-bridging-minds-and-code.md	yes

7 · THE WRITING STYLE, AND THE SEVEN GOALS

NARRATION	IN THE LIBRARY AT THE PIN
stz-functions-as-linguistic-expressions.md	yes
stz-conditional-code-narration.md	yes
stzlist-in-strings-narration.md	yes
stzstring-vizfind-narration.md	yes
stz-function-forms-narration.md	yes

8 · FUNCTIONS AS LINGUISTIC EXPRESSIONS, AND SCOPE

NARRATION	IN THE LIBRARY AT THE PIN
stz-functions-as-linguistic-expressions.md	yes
stz-function-forms-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes

9 · PATTERNS OF THOUGHT: THE SOFTANZUTER, REAXIS AND THE ENGINES

NARRATION	IN THE LIBRARY AT THE PIN
stzregexuter-regex-as-computational-reactive-medium.md	yes
stz-reaxis-reactive-programming-narration.md	yes
stz-managing-deep-lists-narration.md	yes
softanza-reactive-objects-narration.md	yes

10 · PROGRAMMING BECOMES REFINEMENT

NARRATION	IN THE LIBRARY AT THE PIN
stzpolycode-refinement-centered_programming.md	yes

11 · LAW: A LANGUAGE IS DECLARED AND CLOSED, AND WHATEVER SPEAKS IT, PERSON OR MODEL, IS JUDGED BY IT

NARRATION	IN THE LIBRARY AT THE PIN
stznatural-vs-ring-naturallib.md	yes
stz-near-natural-language-programming.md	yes
stz-bridging-minds-and-code.md	yes
stzlistex-pattern-matching-for-lists.md	yes

12 · A LANGUAGE, DECLARED

NARRATION	IN THE LIBRARY AT THE PIN
stz-conditional-code-narration.md	yes
stznatural-vs-ring-naturallib.md	yes
stznatural-narration.md	yes
stz-bridging-minds-and-code.md	yes

13 · NATURAL, AND EXECUTABLE: TWO DIRECTIONS, ONE DISCIPLINE

NARRATION	IN THE LIBRARY AT THE PIN
stznatural-vs-ring-naturallib.md	yes
stz-near-natural-language-programming.md	yes
stznatural-narration.md	yes
stz-bridging-minds-and-code.md	yes

14 · KNOWLEDGE PROGRAMMING

NARRATION	IN THE LIBRARY AT THE PIN
stz-knowledge-programming-narration.md	yes
stz-bridging-minds-and-code.md	yes
stz-rules-as-graph-rules-narration.md	yes

15 · PATTERN LANGUAGES

NARRATION	IN THE LIBRARY AT THE PIN
stznumbrex-pattern-language-for-number.md	yes
stztablex-pattern-langauge-for-tables.md.md	yes
stzlistex-pattern-matching-for-lists.md	yes
stzTimex-A-Pattern-Language-for-Time-in-Softanza.md	yes
stzMatrex-Declarative-Pattern-Matching-for-Matrices.md	yes

16 · SEVEN LANGUAGES, ONE DOOR

NARRATION	IN THE LIBRARY AT THE PIN
stzexcis-polyglot-programming-in-ring.md	yes
stzexterlib-domain-driven-polyglot-programming.md	yes
stzexcis-performance-battle.md	yes
stz-pycode-bridging-ring-and-python-narration.md	yes

17 · LAW: PROGRAMMATIC BY DEFAULT, THE MODEL LAST, ADMISSION OVER GENERATION

NARRATION	IN THE LIBRARY AT THE PIN
stz-agents-that-cannot-hurt-you-narration.md	yes
stz-service-virtualization-code-first-subscribe-later.md	yes
stz-knowledge-programming-narration.md	yes

18 · THE FIVE REGISTERS, AND CONVERSATIONS THAT ASK

NARRATION	IN THE LIBRARY AT THE PIN
stz-knowledge-programming-narration.md	yes
stznatural-narration.md	yes

19 · THE DOMAIN LANGUAGE MODEL

NARRATION	IN THE LIBRARY AT THE PIN
stz-service-virtualization-code-first-subscribe-later.md	yes
stz-knowledge-programming-narration.md	yes

20 · AGENTS THAT CANNOT HURT YOU

NARRATION	IN THE LIBRARY AT THE PIN
stz-agents-that-cannot-hurt-you-narration.md	yes
stz-service-virtualization-code-first-subscribe-later.md	yes

21 · THE COURT: PROMISES, PERCEPTION, SECURITY AND PERFORMANCE

NARRATION	IN THE LIBRARY AT THE PIN
stz-security-ledger-narration.md	yes
stz-governance-lineage-narration.md	yes
stz-perf-governed-loop-narration.md	yes
stz-honest-stopwatch-narration.md	yes
stz-security-drill-narration.md	yes

22 · COMPARED, HONESTLY

NARRATION	IN THE LIBRARY AT THE PIN
stzgraph-vs-other-graph-libraries.md	yes
stzexcis-performance-battle.md	yes
stz-vs-qt-unicode-narration.md	yes
stznatural-vs-ring-naturallib.md	yes
stz-near-natural-language-programming.md	yes

23 · A WORLD, A CONSTELLATION, A PLATFORM, AND THE ARCHITECT

NARRATION	IN THE LIBRARY AT THE PIN
stzApp-building-your-first-app-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes
stz-built-on-real-world-need.md	yes

24 · REHEARSE, THEN COMMIT

NARRATION	IN THE LIBRARY AT THE PIN
stz-service-virtualization-code-first-subscribe-later.md	yes

NARRATION	IN THE LIBRARY AT THE PIN
stz-emulating-the-whole-solution-narration.md	yes
stz-deploying-to-target-sites-narration.md	yes
stz-planning-and-provisioning-a-deployment-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes

25 · YOUR USERS KEEP THEIR MEDIUM

NARRATION	IN THE LIBRARY AT THE PIN
stz-service-virtualization-code-first-subscribe-later.md	yes
stznatural-narration.md	yes
stzApp-building-your-first-app-narration.md	yes

26 · THE ENVIRONMENT

NARRATION	IN THE LIBRARY AT THE PIN
stzApp-building-your-first-app-narration.md	yes
stz-emulating-the-whole-solution-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes

27 · THE ENGINE BENEATH: THREE LAYERS, ONE ENGINE, HARO

NARRATION	IN THE LIBRARY AT THE PIN
stz-engine-senses-narration.md	yes
stz-vs-qt-unicode-narration.md	yes
stz-emulating-the-whole-solution-narration.md	yes
stz-honest-stopwatch-narration.md	yes

28 · A RESTAURANT IN LYON

NARRATION	IN THE LIBRARY AT THE PIN
stz-service-virtualization-code-first-subscribe-later.md	yes
stz-perf-judgment-narration.md	yes
stzApp-building-your-first-app-narration.md	yes

29 · A BANK IN NIAMEY

NARRATION	IN THE LIBRARY AT THE PIN
stzOrgChartDoc-BankUseCase.md	yes
stz-rules-as-graph-rules-narration.md	yes

30 · AN ORGANISATION IN THE FIELD

NARRATION	IN THE LIBRARY AT THE PIN
stz-rules-as-graph-rules-narration.md	yes
stz-governance-lineage-narration.md	yes

31 · WHAT THE PROJECTS SHOW, AND WHAT COMES NEXT

NARRATION	IN THE LIBRARY AT THE PIN
stz-built-on-real-world-need.md	yes
stz-service-virtualization-code-first-subscribe-later.md	yes
stz-getting-paid-and-paying-narration.md	yes
stz-system-dev-to-deploy-narration.md	yes

32 · MANY MINDS ON ONE BODY OF WORK

NARRATION	IN THE LIBRARY AT THE PIN
stz-rules-as-graph-rules-narration.md	yes
stz-governance-lineage-narration.md	yes

33 · THE NEXT LINE: A SOLUTION ON ITS OWN ENGINE

NARRATION	IN THE LIBRARY AT THE PIN
stz-built-on-real-world-need.md	yes
stz-system-dev-to-deploy-narration.md	yes
stz-emulating-the-whole-solution-narration.md	yes
stz-service-virtualization-code-first-subscribe-later.md	yes

APPENDIX G

The book as data, for the agent that reads it

DRAFT

This page has no code to run.

This appendix is for an AI agent that reads the book, and for the person who gives it the book to read. It says what the book offers in a form a machine can take, and what an agent may do with it.

WHAT THERE IS

Files sit beside the reader, and each is regenerated at every build.

- **`book.json`** holds, for each page of the book, a chapter, the preface or an appendix: its number, identifier and slug; its kind; its part; the law it states, if it is a law chapter; the first principle it states, in one sentence; whether it is on the spine; its status; its stage words, built, designed and absent; its guards, each with its path and its assertion count; its promises, which are its blocks of code with the lines they promised, what the last run printed, and the verdict; its narrations; the study's rulings it rests on; its six doors; its sections; its open questions, which are the places the record could not answer; its word count; whether it has a French edition; and where its text is. A chapter that is a draft says so in its status.
- **`book.schema.json`** describes `book.json`, and the build validates the file against it. A build that produces a `book.json` that does not match its schema fails.
- **`llms-book.txt`** is a short text: the rules for an agent, then one line per page with its status, its guards and the count of blocks that kept their promise, and its principle.

- **`llms-full-book.txt`** is every page as text, one after the other, with the verdict of the last run written under each block of code. **`agents/<slug>.md`** is the same, one page per file; the slug is in `book.json`.
- **`status.html`** is the page for a person who builds the book: every chapter with its status, its run, its guards, its words, its open questions and whether it has a French edition. It is in the repository and not on the public site, because it prints the library's file names as they are.

WHAT A PROMISE IS

A block of code that begins with the tag `haro` is run inside the library at the commit named under the chapter's title. The lines that begin with `#-->` are the block's promise: what the block printed when the author wrote it. The judge compares the promise with the run, ignoring spacing, and reading `TRUE` and `FALSE` as 1 and 0. A promise is kept, or it differs, or the block raised, or it stopped. A block that is marked as an exercise is not run, and a block that touches files, the clock or chance says so, and shows the output that was recorded on its date.

An agent that quotes a promise should quote its verdict with it.

WHAT AN AI AGENT MAY DO, AND MAY NOT

- **You may quote any cell.** A table of this book is data. A row that says "absent" is a row you may repeat as absent.
- **You may not turn an absent cell into a present one.** The record will find your sentence and the missing file side by side, and the book says so in chapter 31.
- **You may rely on the stage words.** Built means the code, a guard and a run. Designed means a document and nothing that runs. Absent means that I looked, and that the chapter says how far.
- **You may rely on a promise only with its verdict, and only at the commit it names.** A number that the book prints is a number read from a file or a run, at the date the page gives.

- **A draft is a draft.** A chapter marked as a draft is not the author's final word, and "the author read it" is a status that only the author sets.

THE CHAPTERS, AS THE BUILD READS THEM

The table below is generated at every build from the same data as `book.json`.

#	ID	PART	STATUS	SPINE	BLOCKS KEPT	GUARDS	DOORS
	preface	0	draft	yes	1 of 1	0	0
1	change-arrives-faster-than-trust	1	draft	yes	2 of 2	3	6
2	the-software-crafter	1	draft	yes	9 of 9	5	6
3	the-method-from-first-principles	1	draft	yes	5 of 5	3	6
4	the-platform-in-one-picture	1	draft	yes	8 of 8	5	6
5	law-the-code-adheres-to-the-way-a-person-thinks	2	draft	yes	9 of 9	4	6
6	the-mental-model-and-the-metaphors	2	draft		7 of 7	6	6
7	the-writing-style-and-the-seven-goals	2	draft		9 of 9	10	6
8	functions-as-linguistic-expressions-and-scope	2	draft		13 of 13	5	6
9	patterns-of-thought	2	draft		8 of 8	6	6
10	programming-becomes-refinement	2	draft		9 of 9	3	6
11	law-a-language-is-declared-and-closed	3	draft	yes	11 of 11	7	6
12	a-language-declared	3	draft		8 of 8	5	6
13	natural-and-executable	3	draft		17 of 17	7	6
14	knowledge-programming	3	draft		13 of 13	3	6
15	pattern-languages	3	draft		13 of 13	5	6
16	seven-languages-one-door	3	draft		10 of 10	5	6
17	law-programmatic-by-default	4	draft	yes	8 of 8	6	6
18	the-five-registers	4	draft		15 of 15	4	6
19	the-domain-language-model	4	draft		7 of 7	5	6

#	ID	PART	STATUS	SPINE	BLOCKS KEPT	GUARDS	DOORS
20	agents-that-cannot-hurt-you	4	draft		9 of 9	5	6
21	the-court	4	draft		7 of 7	9	6
22	compared-honestly	4	draft		5 of 5	4	6
23	a-world-a-constellation-a-platform	5	draft		10 of 10	5	6
24	rehearse-then-commit	5	draft		11 of 11	7	6
25	your-users-keep-their-medium	5	draft		11 of 11	6	6
26	the-environment	5	draft		10 of 10	6	6
27	the-engine-beneath	5	draft		9 of 9	4	6
28	a-restaurant-in-lyon	6	draft		5 of 5	5	6
29	a-bank-in-niamey	6	draft		6 of 6	6	6
30	an-organisation-in-the-field	6	draft		7 of 7	6	6
31	what-the-projects-show	6	draft		5 of 5	8	6
32	many-minds-on-one-body-of-work	7	draft	yes	3 of 3	3	6
33	the-next-line	7	draft	yes	3 of 3	3	6
A	the-register-of-innovations		draft		no code	0	0
B	the-fourteen-house-rules		draft		2 of 2	1	0
C	the-twenty-two-verbs-and-the-constitution		draft		no code	0	0
D	the-glossary		draft		no code	0	0
E	the-guards-and-fixtures-cited		draft		no code	0	0

#	ID	PART	STATUS	SPINE	BLOCKS KEPT	GUARDS	DOORS
F	the-narrations-the-book-argues-from		draft		no code	0	0
G	the-book-as-data		draft		no code	0	0
H	the-projects-history		draft		no code	0	0

APPENDIX H

The project's history: from Ring to Haro

DRAFT

This page has no code to run.

This is the one page of the book whose prose names the language Softanza grew up on; the library's own names inside code still carry it, until the library is retired from it. The chapters do not, because the platform is going where that language cannot follow. The history belongs here, told once, with thanks.

WHERE IT BEGAN

Softanza was born on Ring, a small scripting language. I wrote its library in Ring for years. I taught Ring, I promoted it in the field, and I spent nine months writing its book, *Beginning Ring Programming*, published by Apress in 2020. In 2021 I described the library to the Ring community for the first time. In 2022 the first presentation of Softanza called it "a best friend of existing Ring programmers". Ring's simplicity and its multiparadigm expressiveness are two things Softanza keeps, because they were never the problem.

WHAT REAL PROJECTS TAUGHT

Three things, and they are the reasons for the move to Haro.

The limits showed. Real projects showed that Ring was not suited to the challenges of modern applications. The limits are in three places: performance, strong typing and ease of build. The project's own sessions measured them, and the record keeps the rows with their reproductions. Chapter 27 prints the measure of the first: the engine within 2x of a fast peer of the field on every row, which the former runtime never was. Its memory stands beside the former runtime's.

The direction could not be influenced. The direction of a language belongs to its owner, and we could not influence the direction of ours. The project built Ring++ to solve those three pains without leaving Ring, and it did not meet the reception we hoped for.

Governance is a design property. A dependency whose direction you cannot influence, and whose behaviour you cannot fence, fails the due diligence that regulated domains require. Softanza's answer is not independence for its own sake, but independence for its users: an engine written in Zig with no third-party dependency, a virtual machine and a compiler of our own, and commitments the project already runs as tests. An open core. Charters ratified in public. Every decision in a dated record. Every claim linked to its proof.

THE PATH

Ring is where it began. Ring++ is the bridge, under construction. Haro is the destination. Under the library sits the engine in Zig, whichever of the three is above it: every feature is implemented once, in the engine, and chapter 27 tells the rule.

Today the library still runs on its first runtime, version 1.27, and the blocks of this book run on it. Haro uses the syntax of that first runtime as its first facet, so all the code already written keeps working. The library's guards are the test of that promise: Haro has to run them. Chapter 27 tells where the engine stands, with numbers.

THANKS

Softanza says thank you to the language it grew up on, and goes where its users' sovereignty requires.

If this book is useful to your organisation, I work with organisations on its subjects: paper copies in quantity, workshops and training, talks, consulting and the building of solutions. Write to kalidianow@gmail.com.

How these books are offered : https://mayouni.github.io/ayouni/offer/

Free to read and share under CC BY-NC-ND 4.0; code under the library's licence.

https://mayouni.github.io/ayouni/licence/

Declare, then govern.

Born in Africa. Useful to the World!

You use a tool that writes code. You ask it for a function, or for a whole feature, and it answers in seconds. Then comes the question that no tool answers for you: should this code enter my system?

Code has become cheap. Trust has not become cheap. **Change arrives faster than trust.**

WHAT SOFTANZA IS

Softanza is a governed software platform. You declare what you know about a world, in plain text; a court built from that declaration keeps it; and the AI model, where there is one, comes last. It is three things: a library you call from code, an engine beneath it that does the computation, and a language, Haro, made to hold the declarations.

WHO IT IS FOR

The software crafter: whoever declares what they know about a world, and owns what they declare. The programmer, the analyst, the designer, the decision-maker, the educator, and the AI agent, since this book is also published as data a program can read. You do not need to know Softanza. You need to have written code, and to have felt the doubt above.

Every code block in this book was run, and its verdict is printed under it.



MANSOUR AYOUNI

He wrote his first program at ten, with pen and paper, in Regueb, a village in the centre of Tunisia that had no electricity. Programming then took him from Tunisia to Niger and to Canada, building software for law offices, banks and nuclear waste management, and leading teams of three cultures. He designed Softanza and was its first crafter.

AYOUNI

FIRST OF THE AYOUNI BOOKS

Free to read, in every form

mayouni.github.io/ayouni/books/softanza/en/

PROOF COPY · CC BY-NC-ND 4.0