

Nasdanika SDK Roadmap

What Comes Next, and How to Take Part

Plans, modules to try and models to migrate

Pavel Vlasov

roadmap.sdk.nasdanika.org



Nasdanika SDK Roadmap

What Comes Next, and How to Take Part

Pavel Vlasov

Version 2026.11.0, 2026-11-11

Table of Contents

Colophon . . . 1

Preface . . . 3

What it is for . . . 3

How to read it . . . 3

How it is kept . . . 4

What is not here . . . 5

I: The Roadmap . . . 6

1. Direction . . . 7

1.1. One model in the middle . . . 7

1.2. The North Star . . . 8

1.3. How the next step is chosen . . . 8

1.4. Where things stand . . . 9

1.5. What holds . . . 10

1.6. The shape of what follows . . . 10

2. Taking Part . . . 12

2.1. Try it . . . 12

2.2. Take a piece . . . 12

2.3. Sponsor . . . 13

2.4. What an item says . . . 13

2.5. What makes an item graduate . . . 14

3. Working with AI Assistants . . . 15

3.1. What to give an assistant . . . 15

3.2. How to ask . . . 15

3.3. What an assistant is for, and what it is not . . . 16

II: Runtime . . . 18

4. Bindings . . . 19

4.1. Why . . . 19

4.2. The interface . . . 20

4.3. The value now, and then each change . . . 20

4.4. Back pressure is coalescing . . . 21

4.5. The pump sets the pace . . . 22

4.6. Where a change came from . . . 23

4.7. Computed bindings . . . 24

4.8. Bindings that follow each other . . . 25

4.9. A model feature is a binding . . . 26
4.10. Closing . . . 26
4.11. What is not there yet . . . 27
4.12. Prior art . . . 27
5. Waypoints . . . 29
5.1. Why . . . 29
5.2. Lineage . . . 30
5.3. State . . . 32
5.4. Settling is a rule over lineage . . . 32
5.5. Realms . . . 33
5.6. Telemetry . . . 35
5.7. What changed on the way into the SDK . . . 36
5.8. Designed, not built . . . 36
5.9. Open questions . . . 39
6. Widgets . . . 40
6.1. Why . . . 40
6.2. A widget . . . 41
6.3. One view, four renderings . . . 41
6.4. How a kind is found . . . 45
6.5. Text carries its format . . . 47
6.6. Markdown . . . 49
6.7. Surfaces and the terminal . . . 51
6.8. A binding is a value . . . 53
6.9. What is not there yet . . . 53
6.10. Why not something that exists . . . 54
7. The Terminal . . . 55
7.1. Why a terminal . . . 55
7.2. TamboUI as a provider . . . 56
7.3. The way out, in use . . . 58
7.4. Screenshots . . . 60
7.5. What is not there yet . . . 61
8. Nasdanika Commander . . . 63
8.1. Why . . . 63
8.2. Running it . . . 64
8.3. The application is not the terminal . . . 65
8.4. Viewing and editing . . . 67

8.5. Copy, move and delete . . . 70
8.6. A panel is a volume . . . 73
8.7. Through a real terminal . . . 76
8.8. What is not there yet . . . 77
9. Screens Drawn in Draw.io . . . 78
9.1. Why . . . 78
9.2. Draw, run, then bind . . . 79
9.3. What is drawn . . . 80
9.4. How a drawing meets Java . . . 80
9.5. One drawing, several surfaces . . . 81
9.6. Usage against design . . . 82
9.7. Prior art . . . 82
10. Designed and Waiting . . . 83
10.1. Toward the North Star . . . 83
10.2. Seeing and changing a running system . . . 84
10.3. The command line . . . 84
10.4. Documents and diagrams . . . 85
10.5. Books and the body of knowledge . . . 86
III: Models . . . 87
11. Core Nano-models . . . 88
11.1. What a nano-model is . . . 88
11.2. Related . . . 90
11.3. Change . . . 91
11.4. Occupancy . . . 91
11.5. Kind . . . 92
11.6. Enterprise . . . 94
11.7. Document . . . 95
11.8. Example . . . 96
11.9. Role, IAM, Seal and Lifecycle . . . 97
11.10. What is not there yet . . . 97
12. Runtime Models . . . 98
12.1. CLI . . . 98
12.2. Tool . . . 99
12.3. Inference . . . 100
12.4. Expression . . . 100
12.5. Cypher . . . 101

12.6. XPath . . . 101
13. The Model Tower . . . 102
13.1. What the tower is . . . 102
13.2. What changes with the SDK . . . 103
13.3. The floors . . . 104
13.4. One user interface or many . . . 107
13.5. Order . . . 108
14. OpGraph and NSML . . . 109
14.1. OpGraph . . . 109
14.2. NSML . . . 111
15. Technology Models . . . 114
15.1. What migration means . . . 114
15.2. The models . . . 115
15.3. Formats already in the runtime . . . 116
Appendix A: Items . . . 117
A.1. Runtime . . . 117
A.2. Models . . . 119
Appendix B: Revisions . . . 121
Appendix C: Building and Conventions . . . 124
C.1. The repository . . . 124
C.2. Building . . . 124
C.3. The book . . . 125
C.4. Publishing . . . 125
C.5. Conventions . . . 126
C.6. Adding an item . . . 126
C.7. When an item graduates . . . 127
C.8. TamboUI on the module path . . . 127
Appendix D: Glossary . . . 128
Appendix E: The Nasdanika Body of Knowledge . . . 132
E.1. What it holds . . . 132
E.2. How it is laid out . . . 133
E.3. How it is published . . . 133
E.4. How to read it . . . 134
E.5. Why it can be trusted, and for which version . . . 134
About the author . . . 136

Colophon

Nasdanika SDK Roadmap: What Comes Next, and How to Take Part

Edition 2026.11.0, built on 2026-11-11.

This book is published in three formats, and each is rebuilt whenever the roadmap changes:

- a web page, roadmap.sdk.nasdanika.org;
- a PDF, roadmap.sdk.nasdanika.org/nasdanika-roadmap.pdf;
- an EPUB, roadmap.sdk.nasdanika.org/nasdanika-roadmap.epub.

It is also on Leanpub, leanpub.com/nasdanika-sdk-roadmap, as part of the Nasdanika Body of Knowledge. Its source, with the modules, the tests and the models it describes, is at github.com/Nasdanika-SDK/roadmap.

This edition is built on Nasdanika SDK Runtime 2026.11.0. Every code listing in it is a region of a source file or a test in the repository, compiled and tested against that version of the runtime when the book is built, and every terminal screenshot is what a test drew.

Copyright © 2026 Pavel Vlasov.

The text and figures of this book are licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). You may share and adapt them for any purpose, commercial purposes included, provided you give appropriate credit, provide a link to the license, and indicate if you made changes. A suggested credit: "Pavel Vlasov, *Nasdanika SDK Roadmap*, licensed under CC BY 4.0", with a link to github.com/Nasdanika-SDK/roadmap.

The code and the models are licensed under the Eclipse Public License 2.0, the license of the runtime. In addition, you may use the code of this book's listings in your own programs and documentation without asking for permission and without attribution.

Neither license covers the Nasdanika name or logo: CC BY 4.0 grants no trademark rights.

Eclipse, EMF and Xtext are trademarks of the Eclipse Foundation. Java and OpenJDK are

trademarks or registered trademarks of Oracle and its affiliates. OpenTelemetry is a project of the Cloud Native Computing Foundation. Other names may be trademarks of their owners. Their use here identifies the technologies the roadmap builds on and implies no endorsement.

Typeset with Asciidoctor PDF in Noto Serif, Noto Sans and M+ 1mn. Terminal screenshots exported by TamboUI.

Preface

A roadmap is usually a slide: boxes on a timeline, true on the day it was drawn. This one is a repository. It has a book, the one you are reading, and it has code: the modules that may become part of the Nasdanika SDK, the tests that show how to use them, and the models that are to be migrated. The code is built on a released runtime, 2026.11.0, so whatever has a module here can be tried today, and whatever has only a chapter is a statement of intent.

What it is for

To say in one place where Nasdanika is going. Until now the vision and the plans lived in several places: the nasdanika.com site, a growing file of dated decisions, the readmes of fifty-odd model repositories, and design documents for Waypoint, OpGraph and the Nasdanika Semantic Mapping Language. This book gathers what is public in them, and the places it replaces will point here.

To announce intent early, with something to run. A proof of concept in this repository is not a release. It is there so that someone who needs the thing can see what it would be, take it apart, and say what is wrong with it while changing it is still cheap.

To make it possible to take part. Some readers will want an item enough to contribute to it, or to sponsor it. Others will take a piece and build their own solution on it, which the license allows and which is welcome. Chapter 2 says how.

To ground AI assistants. An assistant that has read this book and the repository can build on the roadmap rather than on its guess of what a Nasdanika module looks like. Chapter 3 shows how to ask.

How to read it

Start with Chapter 1, which says what the destination is and how the next step is chosen. After that the book is a reference: go to the item you care about. Appendix A lists every item with its status and the chapter that describes it.

Three labels say how real something is:

- **AVAILABLE**: in a released runtime. This book mentions such things only as the ground the roadmap stands on. The SDK book describes them.
- **SKETCH**: there is code in this repository, built and tested, and it is not settled. Names, packages and behavior may change before it graduates, and it may not graduate at all.
- **PLANNED**: designed, or only intended, and not built.

A chapter about something with code has the same shape each time: why, what it is, how it is used, with listings from the tests, what is not there yet, and how to help.

How it is kept

The roadmap changes as work is done and as ideas arrive, and it is kept by two rules.

The first is that the chapters say what holds now, and Appendix B says when and why it came to hold. A chapter is rewritten when a decision changes it, and the decision gets a dated entry, so nothing changes silently and nobody has to read the history to learn the present.

The second is how the next step is chosen, which I call gradient ascent: from where things stand, take the step with the most value for its time and effort, counting what it makes cheaper later, and then look again from the new position. Chapter 1 has more on it.

The book is built on a runtime release and carries its version, 2026.11.0. The date on the title page says when this copy was built. A module graduates from the roadmap to the runtime by dropping `roadmap` from its names: `org.nasdanika.sdk.roadmap.runtime.widget` becomes `org.nasdanika.sdk.runtime.widget`. When it does, its chapter leaves this book and its tests become the examples of the runtime's books.



A first cut

This edition is the first cut of the book. It was assembled with an AI assistant from the roadmap's earlier files, the nasdanika.com site, the model repositories and the experiments of November 2026. I have reviewed it, and I did not go into the details. That is by design. The intent is generally correct, and for a roadmap that is what has to be right. What it says about the behavior of code is backed by tests that run when the book is built.

> What it says about plans, priorities and reasons is a reading of those sources: right in direction, and not checked line by line.
>
> A section gets its in-depth review at one of two moments: when it is time for it to graduate from the roadmap, or when it is elaborated while it stays in the roadmap. Until then, read a section as a statement of intent.

What is not here

Stories and their backlog are in the stories repository. What is learned about third-party technologies along the way, such as how TamboUI exports a screen, goes to the Nasdanika Field Notes, where it is proved by a test that needs no Nasdanika code.

Part I: The Roadmap

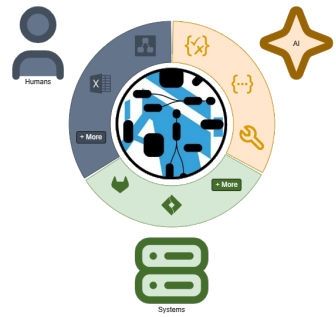
Where Nasdanika is going and how the next step is chosen, what a reader can do with a roadmap that has code in it, and how to work with an AI assistant that has read it.

Chapter 1. Direction

1.1. One model in the middle

Nasdanika starts from one picture. In the middle is the model: not one large model but a federation of small ones, each small enough to read in one sitting, formal enough for tools and agents to work on, and composable with the others the way Java modules compose through Maven. A small model is cheap to write and cheap to publish, and that is what makes "one model at a time" a way to adopt modeling rather than a slogan.

Around the model are three audiences, and each meets it through its own surface.



People write and read in the tools they already use: Draw.io diagrams, spreadsheets, Markdown, generated sites and books, a command line, a terminal. They get a user interface for a task, or a group of tasks, rather than one portal for everything.

AI agents do not get raw files. They get schemas and tool definitions generated from the metamodels, and the slice of the model that a task needs, so that they stay focused and bounded.

Systems are where the work already happens: Git hosting, issue trackers, build tools, a running JVM. Connectors load what is in them as models and cross-reference it. What is loaded can be saved as a snapshot and worked on later, away from the source: enriched in several passes, examined after access has expired, and compared with the next snapshot.

The model is kept in Git, which gives versions and distributed ownership file by file. Change recording and telemetry, kept as models beside the model, give the finer grain that Git alone does not have: which feature changed, in which unit of work, caused by what.

1.2. The North Star

The destination is agentic work on bounded, typed models: inference that runs in the JVM, with tool calls recorded as models, state in Git, the safety of a transaction, and the same behavior on a laptop, in a container and in a build.

The smallest demonstration that would be believed is one agentic workflow whose state is in Git and can be rolled back. The candidate is the dispute of a card transaction in the Nasdanika International Banking System, the reference application that grows with the platform.

The North Star is what keeps gradient ascent from stopping on a local maximum. A step can be valuable and cheap and still lead away from it.

1.3. How the next step is chosen

Gradient ascent. A plan with dates for the next two years would be wrong within a month, since each thing that is built changes what the next thing costs and what it is worth. So the plan is a position and a rule. From where things stand, take the step with the most value for its time and effort, counting what it makes cheaper afterwards. Then measure again from the new position. Items in this book have a status and what they wait for, not a date, except the next release.

Socialize before inference. The capabilities that need no inference are shown first: observability, insight into a running JVM, diagrams, documentation that cannot drift from the code. They have value on their own, to developers and architects, and they are the ground that inference will stand on. The runtime released so far is the result of this rule.

An increment says what it adds. Each increment states the capability it adds, the demonstration that shows it, what it pulls in, and that it builds green from a clean clone. The chapters of this book with code follow that shape.

Salvage, do not port. The older Nasdanika has a great deal of code. A module comes from it into the SDK as a small library, and its integrations come as adapters when something needs them. Nothing is moved because it exists.

Demonstrate on one example. New capabilities are shown in the Nasdanika International Banking System where a bank would plausibly need them, and in the module's own book where it would not.

1.4. Where things stand

The Nasdanika SDK Runtime 2026.11.0 is released to Maven Central. Table 1 lists what is in it, as the starting position of this roadmap. The SDK book and the Nasdanika Zephyr Books, one for a module or a small group of modules, describe it.

Table 1. What the runtime has, as of 2026.11.0

Area	Modules
Foundation	`common`: flows on the JDK (`Pump`, `Transformer`), the capability loader, telemetry per unit of work, resource sets assembled from services
Files and serving	`volume`: the `Volume` SPI, content, an EMF URI handler over volumes, HTTP and WebDAV handlers
Testing with traces	`harness`, `playwright`: a server in the trace of its test, browser sessions with steps, screenshots, narration and video
Persistence	`emf-json`: JSON and YAML resources
Command line	`picocli`, `cli`, `bootstrap`, `cli-launcher`: commands contributed by modules and wired by generic type, and a distribution
DSL	`groovy`: a Groovy DSL for any model
Draw.io	`drawio`, `drawio-playwright`: a Java API for diagrams, and rendering with the viewer of draw.io
Models	core, file, telemetry, typescript, meta, jvm, html with Bootstrap, react with React Flow, diagram with ELK, presentation, drawio, svg

What grew first is the half that needs no inference: telemetry, volumes, the command line, documentation, diagrams and browser automation. Inference, its providers, MCP, the change model, messaging and Cypher are not in the runtime yet. They are the path to the North Star, and Chapter 10 lists them.

1.5. What holds

Some decisions are settled and shape everything in this book. The SDK book argues them; here they are in one line each.

- **One identity.** A module's path gives its site page, namespace URI, Java module, package, Maven coordinates, logger and instrumentation scope.
- **Pure Nasdanika** is Java, EMF core and the OpenTelemetry API. Anything else earns its place module by module.
- **The JDK first.** `Flow`, virtual threads and the JDK's HTTP client and server, with no reactive library under them.
- **Services before capabilities.** A lookup by type is a Java service. The capability framework is for requirements that are objects, and for resolution that chains.
- **Telemetry per unit of work.** Spans for operations, log records for what happens in them, and one `OpenTelemetry` per unit of work carried in the context, never a global one.
- **Three planes.** Data in volumes, semantics in resource sets, representation in views. Reads go up through volumes and writes go down through commands.
- **A polymorphic user interface.** A view is chosen by the class of what it shows and by the format asked for, and an agent is a user too.
- **Documentation is built.** Books are AsciiDoc in the module they describe, their listings are regions of tests, and a warning fails the build.

1.6. The shape of what follows

The rest of the book has two parts.

Runtime is what comes next in the runtime itself. The order of its chapters is the order of dependency: bindings and waypoints go into `common`, widgets are built on bindings, the terminal on widgets, and the commander on the terminal and on volumes. Screens drawn in Draw.io, and the list of what is designed and waiting, close the part.

Models is the larger and the slower part. The core model gains nano-models, small aspects that many models share. The models of the runtime itself, such as the command line and inference, come next. Then the model tower, the "business" models that build on each

other floor by floor, and the technology models that load what is in other systems.

The runtime is released quarterly. The next release is planned for February 2027, tentatively.

Chapter 2. Taking Part

A roadmap that is a repository can be used, and not only read. This chapter says what a reader can do with it, from the least commitment to the most.

2.1. Try it

Everything with a module here builds with one command, on Java 25 and Maven, against the released runtime from Maven Central:

```
git clone https://github.com/Nasdanika-SDK/roadmap
cd roadmap
mvn verify
```

The build compiles the modules, runs their tests, generates the nano-models from Xcore, and renders this book with the listings and screenshots the tests produced. Appendix C has the details.

Some things can be used as they are. Nasdanika Commander, in Chapter 8, is a file manager for a terminal, and after the build it runs with `java -jar`.

2.2. Take a piece

The code is under the Eclipse Public License 2.0, and the text under CC BY 4.0. Taking a module, a class or an idea and building your own solution on it is a use the roadmap is meant for. You do not have to wait for an item to graduate, and you do not have to agree with where Nasdanika takes it afterwards.

Two cautions. What is here is a sketch, so its names and behavior will change, and a copy will not follow. And a module's package has `roadmap` in it for that reason: nothing here is published to Maven Central, so that nobody depends on a version of something that was never meant to be stable. === Contribute

A contribution to an item is a pull request to this repository, in the shape the repository already has:

- **code in the module the item belongs to,** or a new module beside it, with a `module-info.java`, depending on the runtime's modules through its bill of materials;
- **tests that show how the code is used,** since they are the documentation: a test is named for what it shows, and the regions a chapter would quote are tagged;
- **the chapter,** changed to say what now holds, with a dated entry in Appendix B when a decision changed.

A model is contributed the same way: an Xcore file with a header that says what the model is and why, in `models/`, and a row in the chapter that lists it. A model does not have to generate code to be worth having here. A sketch that can be read is the first step, and for many models it is the step that settles the words.

2.3. Sponsor

Development is a continuum. Building something is one part of it, and using it, testing it and polishing it until it can be relied on is another, usually the longer one. A sketch in this repository has had the first part and little of the second.

Sponsoring an item means putting resources of your own into moving it along that continuum. Most often those are your people and your cases: a team that takes what the author, or somebody helping him, has built, uses it on real work, tests it where the author's tests do not reach, and polishes what it finds rough. It does not have to mean paying anyone.

Some items are large, and some matter to one organization long before they would reach the top of the queue by value for effort. Sponsoring is a way to move such an item: the work is done in the open, in this repository, under the same licenses, and what the sponsor gets is the item sooner and shaped by their case. The way to start is an issue that names the item and says what you can bring to it.

2.4. What an item says

Each item in this book states the same things, so that items can be compared and so that someone who wants to help knows where to begin.

Capability

What a user will be able to do that they cannot do now.

Demonstration

The example that shows it, in a test or in the Nasdanika International Banking System.

Status

SKETCH or PLANNED, and where the code is if there is any.

Depends on

What has to exist first, in the runtime or in this roadmap.

Value and effort

Why it is worth doing and how large it is, in words. These decide the order.

How to help

What a contributor or a sponsor could take on.

Appendix A lists the items. Later the items may become a model themselves, rendered into this book through the widget module, with the tables and the dependency diagrams generated. The first step toward that is in Chapter 6: the table of runtime items there is rendered by the code it describes.

2.5. What makes an item graduate

A module moves from this repository to the runtime, and to Maven Central, when three things hold.

1. **Its identity is settled.** Its names, and for a model its namespace URI, are the ones it will keep. Maven Central is permanent, so a name published there is a name forever.
2. **It is documented.** It has a book, or a chapter of one, whose listings are its tests.
3. **It has a consumer that is not itself.** Another module, an example in the reference application, or a user who asked for it.

Until then it stays here, where it can change.

Chapter 3. Working with AI Assistants

Most of what is in this repository was built with an AI assistant, and a reader is likely to use one to build on it. An assistant is only as good as what it has read. Asked to "build a file manager for the terminal with Nasdanika", it will produce something plausible from its training, and Nasdanika's modules will not be in it, or will be there as they were in an older line of the code. Given this roadmap, it has the designs, the code and the tests to work from.

3.1. What to give an assistant

The repository, not the rendered book. The AsciiDoc sources are a fraction of the size of the HTML, and they lead to the code: every listing in a chapter is an `include` that names a file and a tagged region. An assistant that can read files finds its way as an engineer would.

1. `AGENTS.md` at the root says how the repository is laid out and what its conventions are.
2. The root `pom.xml` lists the modules, each with a name and a description.
3. `book/src/docs/asciidoc/chapters` has a chapter per item: the why, the design, and what is not built.
4. A module's tests show how it is used, one behavior per test.

The runtime's books, for the ground under it. The roadmap is built on the released runtime and does not repeat its documentation. The SDK book and the Zephyr Books are published with each release, and their sources are in the runtime repository.

3.2. How to ask

Name the item and where it is, say what is yours, and say what done looks like. Some requests that work, with the paths as they are in this repository:

> Build me a commander for the entries of a zip file, based on the Nasdanika SDK Roadmap, chapter "Nasdanika Commander" and `runtime/commander`. One panel over a `Volume` for the zip file and one over the current directory. Add tests in the

> style of `CommanderTests`: keys pressed, the screen read from a buffer.

> Using `runtime/widget` in the Nasdanika SDK Roadmap, write a view of my `Invoice` class with a section, a table of line items and a total. Render it as GitHub Flavored Markdown and on a 100 by 30 terminal, as `RoadmapViewTests` in `tests/widgets` does, and write both renderings to `target`.

> Add a `Tree` kind to the widget module for `TerminalContext`, following how `Table` is done in `TerminalKinds`. Rows are read only when they are shown. Do not add it for Markdown: a view should find that it is missing and use a nested list.

> Sketch a nano-model for approvals in the style of `models/core` in the Nasdanika SDK Roadmap: an abstract class, a definition, a catalog and a reference, with a header comment that says what it is for, and a test that instantiates it.

A request in that form gives the assistant a pattern to follow and a way to know when it has followed it.

3.3. What an assistant is for, and what it is not

The division of work that has held up is this.

The assistant drafts. It writes the code, the tests and the first text, and it reads the whole roadmap each time, which a person cannot do. Keeping a plan with many connected ideas consistent is the kind of work it is good at.

The build judges behavior. A claim about what code does is worth what its test is worth. That is why the listings in this book are regions of tests, and why a warning fails the build.

A person judges meaning. Whether a name is the right word, whether a model says what the business means, whether an item is worth doing: no test answers those. The same holds for subject matter. A model of a regulation drafted by an assistant looks as finished as one reviewed by a specialist, and it is not, so this book labels what a specialist has not reviewed.

Three habits help.

- **Ask for the source of a statement.** The roadmap marks what was read in code, what was run, and what is from memory. Ask an assistant to keep doing that.
- **Run before you bind.** For a user interface, look at it on a screen with nothing connected, then connect it. Chapter 9 is that way of working turned into a tool.
- **Keep decisions dated.** When an assistant changes a design, have it record what changed and why, as Appendix B does. A silent rewrite is how a plan loses its reasons.

Part II: Runtime

What is on the roadmap for the Nasdanika SDK Runtime. The first five chapters have modules in this repository, built on the released runtime and tested with every build: bindings, waypoints, widgets, the terminal and Nasdanika Commander. The last two are designs without code yet.

Chapter 4. Bindings

Capability	A value that can be read now and followed as it changes, the same type for a static document, a live screen, an input field and a wire between two computations.
Demonstration	`BindingTests` in `runtime/common`, and the views of Chapter 6, which take a binding wherever they take a value.
Status	**SKETCH** in `runtime/common`, package `org.nasdanika.sdk.roadmap.runtime.common.flow`, beside where `Flows` and `Pump` are in the runtime.
Depends on	The runtime's `common` module: `Pump`, and EMF core for the bridge.
Value and effort	High for small. Widgets, the terminal, live telemetry and executable graphs all need it, and it is a few classes.
How to help	Try it under load and from several threads, and say where the rules below surprise you. List bindings are the next piece, and they are open.

4.1. Why

A user interface shows values that change, and takes values from a user. JFace and EMF have databinding for that, built on observables. The runtime already has what observables are made of: `Flow` publishers and subscribers, and `Pump`, a queue that runs one task at a time. So a binding here is a small thing on top of them, not a framework beside them.

Bindings are not only for user interfaces. A static format such as Markdown needs the value at the time it renders. Live telemetry, server-sent events and notifications to an agent need the value as it changes. And two computations wired to each other need the latest value of the other, as cells of a spreadsheet do. One type serves all of them.

4.2. The interface

Listing 1. A binding is a supplier, a publisher and closeable

```
public interface Binding<T> extends Supplier<T>, Flow.Publisher<T>, Closeable {

    /**
     * @return true once {@link #close()} was called, or the binding this one follows ended
     */
    boolean isClosed();

}
```

Each supertype is a decision.

`Supplier`, so the current value is `get()`. Wherever code takes a supplier of a value, a table column or a lambda, it takes a binding, and a format that renders once calls `get()` and never subscribes.

`Flow.Publisher`, so a binding is followed by subscribing, and the operators of `Flows` apply to it. The first item is the value at subscription, and every item after it is a change.

`Closeable`, the runtime's own, which declares no checked exception. A binding may hold a subscription to another binding or an adapter on a model object, and closing releases it.

4.3. The value now, and then each change

Listing 2. The first item is the value at subscription

```
ValueBinding<String> account = Bindings.value("Checking");

Collector<String> collector = new Collector<>(Long.MAX_VALUE);
account.subscribe(collector); ①
account.set("Savings");
account.set("Savings"); ②

assertThat(account.get()).isEqualTo("Savings"); ③
assertThat(collector.items).containsExactly("Checking", "Savings");
```

① The collector asks for everything. It receives "Checking" at once.

② A value equal to the current one changes nothing, and nobody is told.

③ `get()` is the value now, with no subscription.

Emitting the current value on subscription removes a race. With a separate "read the value" and "start listening", a change between the two calls is lost. Here there is no gap: the first item is the value at the moment of subscription. Kotlin's `StateFlow` and RxJava's `BehaviorSubject` do the same.

A value is never null, because `Flow` forbids null items. A value that can be absent is a binding of `Optional`, as the EMF bridge below shows.

4.4. Back pressure is coalescing

A reactive stream protects a slow subscriber by holding its source back. A binding cannot do that. An EMF notification is delivered inside the setter, so holding it back would freeze whatever is changing the model, and dropping it would lose a change.

A binding does neither. It keeps the latest value, and a subscriber that has demand gets the value as it is then.

Listing 3. A slow subscriber gets the latest value, not a backlog

```
ValueBinding<Integer> progress = Bindings.value(0);

Collector<Integer> frame = new Collector<>(1); ①
progress.subscribe(frame);
for (int done = 1; done <= 1000; ++done) {
    progress.set(done); ②
}
assertThat(frame.items).containsExactly(0);

frame.subscription.request(1); ③
assertThat(frame.items).containsExactly(0, 1000);
```

① A subscriber that takes one value at a time, as a frame of a screen does.

② The source changes a thousand times. Each change marks the subscriber's slot and returns.

③ The subscriber asks again and gets the value as it is now.

A source that changes every millisecond and a target that looks every three seconds cost one flag written per change and one delivery per look. The target sets the pace, and neither the binding nor a widget knows anything about time.

The whole mechanism is one method, run for a subscriber whenever something may have changed for it:

Listing 4. Serving a subscriber

```
private void step() {
    if (done || cancelled) {
        return;
    }
    Throwable error = failure;
    if (error != null) {
        end();
        subscriber.onError(error);
        return;
    }
    if (requested.get() > 0 && dirty.compareAndSet(true, false)) {
        T value;
        try {
            value = Objects.requireNonNull(get(), "A binding's value is never null");
        } catch (RuntimeException e) {
            end();
            subscriber.onError(e);
            return;
        }
        if (!value.equals(last)) { ①
            last = value;
            if (requested.get() != Long.MAX_VALUE) {
                requested.decrementAndGet();
            }
            subscriber.onNext(value);
        }
    }
    if (completing && !done && !cancelled) {
        end();
        subscriber.onComplete();
    }
}
```

① A subscriber is not told about a value equal to the last one it received.

4.5. The pump sets the pace

A binding delivers through an executor. Without one, a subscriber is called on the thread that made the change. With a `Pump`, every delivery is a task of the pump, and nothing is delivered until somebody pumps.

Listing 5. On a pump, changes wait for whoever drains it

```
Pump pump = new Pump();
ValueBinding<Integer> progress = Bindings.value(0, pump); ①

Collector<Integer> collector = new Collector<>(Long.MAX_VALUE);
progress.subscribe(collector);
progress.set(1);
progress.set(2);
progress.set(3);
assertThat(collector.items).isEmpty(); ②

pump.settle(); ③
assertThat(collector.items).containsExactly(3);
```

① The binding delivers through the pump.

② Three changes, and nothing has been delivered: no task has run.

③ The pump is drained, as a terminal's frame loop would drain it, and the subscriber gets one value, the latest.

This is what JFace calls a realm: one thread of access to what is shown. A screen that drains the pump once per frame sees a consistent state on each frame, and no widget needs a lock.

4.6. Where a change came from

A task of a pump runs in the OpenTelemetry context that was current when it was enqueued. A binding inherits that for nothing.

Listing 6. A change arrives in the context it was made in

```
static final ContextKey<String> CONTROL = ContextKey.named("control");

@Test
void aChangeIsDeliveredInTheContextItWasMadeIn() {
    Pump pump = new Pump();
    ValueBinding<String> amount = Bindings.value("0", pump);

    List<String> seen = new ArrayList<>();
    amount.subscribe(each(value -> seen.add(value + " from " + Context.current().get(CONTROL))));
①
    pump.settle();

    try (Scope scope = Context.current().with(CONTROL, "transfer/amount").makeCurrent()) {
        amount.set("100"); ②
```

```
    }
    pump.settle(); ③

    assertThat(seen).containsExactly("0 from null", "100 from transfer/amount");
}
```

① The subscriber records each value and what the context says about its origin.

② The change is made inside a scope, as a key typed into the amount field of a transfer screen would be.

③ The delivery happens later, outside the scope, and still sees it.

With a span per screen and a span or a log record per control, a change in a model can be traced back to the input and the screen that caused it, with no field added for that purpose. Usage can then be compared with design: the paths that a screen flow drew, against the paths people took.

Two limits should be said plainly. When several changes are coalesced into one delivery, the delivery runs in the context of the first of them. And telemetry of a binding records that a value changed and where, and should not record the value by default, since what a person types may be a name, an account number or a password. A sanitizer between values and telemetry is PLANNED, in Chapter 10.

4.7. Computed bindings

The operators of `Flows` return a plain publisher, which has no `get()`. `Bindings` maps both sides.

Listing 7. A converter

```
ValueBinding<Integer> cents = Bindings.value(1999);
Binding<String> label = Bindings.map(cents, value -> "$%d.%02d".formatted(value / 100, value %
100));

assertThat(label.get()).isEqualTo("$19.99"); ①

Collector<String> collector = new Collector<>(Long.MAX_VALUE);
label.subscribe(collector);
cents.set(2500);
assertThat(collector.items).containsExactly("$19.99", "$25.00"); ②

label.close(); ③
assertThat(collector.completed).isTrue();
```

```
assertThat(cents.isClosed()).isFalse();
assertThat(cents.getSubscriberCount()).isZero();
```

① The value now, mapped.

② A subscriber follows the mapped value.

③ Closing the mapped binding stops it following its source. It does not close the source, which it does not own.

Listing 8. A value computed from two others

```
ValueBinding<Integer> quantity = Bindings.value(2);
ValueBinding<Integer> price = Bindings.value(5);
Binding<Integer> total = Bindings.combine(quantity, price, (q, p) -> q * p);

Collector<Integer> collector = new Collector<>(Long.MAX_VALUE);
total.subscribe(collector);
quantity.set(3);
price.set(10);

assertThat(collector.items).containsExactly(10, 15, 30);
```

A computed binding does not store its value. It computes it from its sources when asked, and tells its subscribers to ask again when a source changes. `Bindings.derive` takes any number of sources, which is a spreadsheet cell: a formula and the cells it reads.

4.8. Bindings that follow each other

Two values can be bound in both directions, and the rule that an equal value changes nothing is what lets them settle.

Listing 9. Two bindings settle

```
ValueBinding<Integer> celsius = Bindings.value(0);
ValueBinding<Integer> kelvin = Bindings.value(273);
celsius.subscribe(each(value -> kelvin.set(value + 273)));
kelvin.subscribe(each(value -> celsius.set(value - 273))); ①

celsius.set(100);
assertThat(kelvin.get()).isEqualTo(373);

kelvin.set(0);
assertThat(celsius.get()).isEqualTo(-273);
```

① Each binding sets the other. A round trip arrives back with the value already there, and

stops.

A loop that converges settles this way. A loop that does not converge would run forever, and it needs a guard, as a spreadsheet refuses a circular reference. Chapter 5 shows where that guard comes from: a value that carries its history can tell that it has been here before.

4.9. A model feature is a binding

Listing 10. The bridge from EMF notifications

```
EClass account = EcoreFactory.eINSTANCE.createEClass(); ①
account.setName("Account");

Binding<Optional<String>> name = Bindings.of(account, EcorePackage.Literals.ENAMED_ELEMENT__NAME,
String.class);
Collector<Optional<String>> collector = new Collector<>(Long.MAX_VALUE);
name.subscribe(collector);

account.setName("CheckingAccount"); ②
account.setName(null);

assertThat(collector.items).containsExactly(Optional.of("Account"), Optional.of(
"CheckingAccount"), Optional.empty());

name.close(); ③
assertThat(account.eAdapters()).isEmpty();
```

① Any EMF object will do. This one is an `EClass`, whose name is a feature like any other.

② A setter notifies, and the binding marks its subscribers. The setter is never held back.

③ Closing removes the adapter from the object.

A many-valued feature is bound as an unmodifiable copy of its list. A subscriber is told that the list changed and reads it again, which is what a screen does on each frame anyway.

4.10. Closing

Listing 11. Closing ends subscribers and keeps the last value

```
ValueBinding<String> status = Bindings.value("running");
Collector<String> collector = new Collector<>(Long.MAX_VALUE);
status.subscribe(collector);
```

```
status.close();
status.close(); ①

assertThat(collector.completed).isTrue();
assertThat(status.get()).isEqualTo("running"); ②
assertThatThrownBy(() -> status.set("done")).isInstanceOf(IllegalStateException.class);

Collector<String> late = new Collector<>(Long.MAX_VALUE);
status.subscribe(late); ③
assertThat(late.items).containsExactly("running");
assertThat(late.completed).isTrue();
```

① Closing twice is closing once.

② The last value stays readable, so a session that has ended can still be rendered.

③ A subscriber that arrives late gets the last value and then the end.

A subscriber that cancels leaves the binding open for the others. The owner of a binding closes it, as the owner of any resource does.

4.11. What is not there yet

- PLANNED **List bindings.** A binding of a list tells that the list changed, not what changed in it. A list binding with a second publisher of changes, whose first item is a reset with the whole list, is designed and not built. It matters for targets that patch what they show, such as a page in a browser.
- PLANNED **Input written through a transaction.** An input field's state is a binding already. Writing it to a model should go through a command inside a model transaction: recorded, validated, and rolled back on cancel.
- PLANNED **Telemetry of bindings,** with spans per screen and per control, and the sanitizer for values.
- PLANNED **Bindings over the wire:** server-sent events for a browser, and notifications for an agent.

4.12. Prior art

JFace and EMF databinding: observables, realms, update strategies and validators. Kotlin's `StateFlow` and RxJava's `BehaviorSubject`: the current value on subscription, keeping only the latest. Spreadsheets and dataflow tools, for cells that follow other cells. Service Component

Architecture, for properties and wires between components, which bindings take one step further: a wired property keeps following its source.

Chapter 5. Waypoints

Capability	The history of an execution as it advances through a structure: where it is, how it got there, what it carries, and which span covers it. Rollback, retry, join and settling are then questions asked of that history.
Demonstration	`WaypointTests` in `runtime/common`.
Status	**SKETCH** in `runtime/common`, package `org.nasdanika.sdk.roadmap.runtime.common.waypoint`. Persistence in Git, human-in-the-loop and the mapping engine are **PLANNED**.
Depends on	The runtime's `common` module: `Pump` and the OpenTelemetry API.
Value and effort	High for medium. It is the carrier that executable graphs, agents and mapping need, and its core is small.
How to help	Port a traversal you have, of a graph or a file tree or a model, onto waypoints and say what was awkward. The Git backend is a self-contained piece of work for someone who knows JGit.

Waypoint was designed as a module of the older Nasdanika, in a repository of its own. This chapter replaces that design document, and the repository is to be archived with a note that points here. What changed on the way into the SDK is in Section 5.7.

5.1. Why

Many things advance through a structure and need to remember how they got where they are.

Structure	Where execution is	What it carries
An executable graph	A node or a connection	Named variables
A file tree walk	A path	A running total

Structure	Where execution is	What it carries
A model traversal	A model element and the reference followed	A resource set
A compiler pass	A syntax node	A symbol table
A state machine	A state or a transition	Variables
A mapping from one model to another	A source element and the rule applied to it	The target under construction
An agent's work	A step	What it knows so far

Workflow engines, state graphs for agents and compilers each have their own answer, bound to their own structure. A waypoint is the part they share, with the structure left as a type parameter, so one set of words serves all of them: waypoint, commit, merge, state, realm, span.

The words are Git's on purpose. A waypoint is a commit: it is never changed, it knows its parents, a step forward makes a new one, and two lines of work join in a merge. Someone who knows Git knows what the methods do.

5.2. Lineage

Listing 12. A waypoint

```
public interface Waypoint<E> {

    /**
     * The element this waypoint records the execution at.
     */
    E getElement();

    /**
     * The waypoints this one was reached from, in creation order: none for a root, one for a
step
     * forward, several for a join.
     */
    List<? extends Waypoint<E>> getParents();

    /**
     * A step forward: a waypoint at the next element with this one as its only parent.
     */
    Waypoint<E> commit(E next);
```

```
    /**
     * A join: a waypoint at the next element with this one as its first parent and the others
after it.
     */
    Waypoint<E> merge(E next, Iterable<? extends Waypoint<E>> additionalParents);
}
```

Listing 13. An execution is a graph of waypoints

```
Waypoint<String> order = Waypoints.root("order");
Waypoint<String> validate = order.commit("validate"); ①
Waypoint<String> price = validate.commit("price"); ②
Waypoint<String> stock = validate.commit("stock");
Waypoint<String> ship = price.merge("ship", List.of(stock)); ③

assertThat(ship.getParents()).isEqualTo(List.of(price, stock));
assertThat(ship.lineage().map(Waypoint::getElement))
        .containsExactly("ship", "price", "stock", "validate", "order"); ④
assertThat(validate.getParents()).isEqualTo(List.of(order)); ⑤
```

① A step forward: a new waypoint with this one as its parent.

② Two steps from one waypoint are a fork. Nothing declares it.

③ A join: a waypoint with two parents.

④ The lineage, nearest first, each waypoint once.

⑤ Advancing changed nothing behind it.

The graph only grows. That is what makes the rest simple.

Listing 14. A retry is a sibling

```
StateWaypoint<String, String> validate = Waypoints.root("order", "received").commit("validate",
"valid");
StateWaypoint<String, String> failed = validate.commit("price", "timeout");

StateWaypoint<String, String> retry = validate.commit("price", "priced"); ①

assertThat(retry.getParents()).isEqualTo(List.of(validate));
assertThat(failed.getState()).isEqualTo("timeout"); ②
assertThat(validate.getState()).isEqualTo("valid");
```

① A retry does not undo the failed attempt. It steps again from the last waypoint that was good.

② The failed attempt stays in the graph, with its state, for whoever asks what happened.

Failure handling is deliberately not part of the interface. An engine that retries, one that

compensates and one that gives up all build on the same two moves: find the waypoint to go back to, and commit from it.

5.3. State

A state waypoint carries a value. The value belongs to the waypoint and is never changed: a transformation makes a new waypoint with a new state, which may be of another type.

Listing 15. A join computes its state from what it joins

```
StateWaypoint<String, String> order = Waypoints.root("order", "3 widgets");
StateWaypoint<String, Integer> price = order.commit("price", 120);
StateWaypoint<String, Boolean> stock = order.commit("stock", true);

StateWaypoint<String, String> ship = price.map(
        "ship",
        joining -> "charge " + joining.get(0).getState() + ", in stock: " + joining.get(1
).getState(), ①
        List.of(stock));

assertThat(ship.getState()).isEqualTo("charge 120, in stock: true");
assertThat(ship.getParents()).isEqualTo(List.of(price, stock));
```

① The mapper gets the joining waypoints, this one first, and reads their states.

5.4. Settling is a rule over lineage

Chapter 4 ended with a loop of values that needs to know when to stop. A raw value cannot tell. A value that carries its lineage can: it has settled when it comes back to where it has been with the state it had there.

Listing 16. Settled, by lineage

```
static boolean settled(StateWaypoint<String, Integer> waypoint) {
    return waypoint.lineage()
            .skip(1) ①
            .filter(earlier -> earlier.getElement().equals(waypoint.getElement()))
            .findFirst() ②
            .map(earlier -> earlier.facet(StateWaypoint.class).orElseThrow().getState().equals
(waypoint.getState()))
            .orElse(false);
}

@Test
void settlingIsARuleOverLineage() {
```

```
    Map<String, String> ring = Map.of("a", "b", "b", "c", "c", "a"); ③

    StateWaypoint<String, Integer> waypoint = Waypoints.root("a", 64);
    while (!settled(waypoint)) {
        waypoint = waypoint.commit(ring.get(waypoint.getElement()), waypoint.getState() / 2); ④
    }

    assertThat(waypoint.getElement()).isEqualTo("b");
    assertThat(waypoint.getState()).isZero();
    assertThat(waypoint.visits("b")).isEqualTo(4); ⑤
    assertThat(waypoint.lineage()).hasSize(11);
}
```

① Skip this waypoint itself.

② The last time the execution was at this element.

③ A ring of three nodes.

④ Each step halves the value and moves on.

⑤ The fourth visit to "b" found what the third had left.

Other rules read the same history: stop when a change is smaller than a tolerance, stop on the tenth pass, stop when a budget is spent. None of them is in the value, and none needs a field added to it.

5.5. Realms

Some state cannot be passed by value: a resource set, a connection, a model that many steps change. A realm is a space of such state that is reached only through commands, so that its rules hold: one command at a time, or a transaction around each. It is the idea of JFace's `Realm`, of EMF's command stack and of an event loop.

Listing 17. A realm

```
public interface Realm<S> {

    /**
     * Runs a command against the guarded state and returns when it completes.
     */
    void execute(Consumer<? super S> command);

    /**
     * Computes a value from the guarded state.
     */
    <T> T apply(Function<? super S, ? extends T> command);
```

```
    /**
     * This realm seen as a realm of another state: a lens, not a snapshot. The adapter runs
inside
     * this realm on every access, so the view keeps this realm's rules.
     */
    default <T> Realm<T> adapt(Function<? super S, ? extends T> adapter) {
        Realm<S> self = this;
        return new Realm<>() {

            @Override
            public void execute(Consumer<? super T> command) {
                self.execute(state -> command.accept(adapter.apply(state)));
            }

            @Override
            public <R> R apply(Function<? super T, ? extends R> command) {
                return self.apply(state -> command.apply(adapter.apply(state)));
            }

        };
    }

}
```

The runtime already has a realm, though it does not call it one: `Pump` runs one task at a time.

Listing 18. A realm on a pump

```
static final ContextKey<String> SCREEN = ContextKey.named("screen");

@Test
void aRealmOnAPumpRunsOneCommandAtATimeInTheContextItWasIssuedIn() {
    Pump pump = new Pump();
    List<String> ledger = new ArrayList<>(); ①
    PumpRealm<List<String>> realm = Realms.pump(ledger, pump);

    RealmWaypoint<String, List<String>> debit = Waypoints.realm("transfer", realm).commit(
"debit");
    try (Scope scope = Context.current().with(SCREEN, "transfer").makeCurrent()) {
        debit.execute(entries -> entries.add("debit 100 from " + Context.current().get(SCREEN)));
②
    }
    RealmWaypoint<String, List<String>> credit = debit.commit("credit");
    credit.execute(entries -> entries.add("credit 100"));

    int entries = credit.apply(List::size);
    assertThat(entries).isEqualTo(2);
    assertThat(ledger).containsExactly("debit 100 from transfer", "credit 100");

    CompletableFuture<Integer> size = realm.submit(List::size).toCompletableFuture(); ③
    assertThat(size).isNotDone();
```

```
    pump.settle();
    assertThat(size).isCompletedWithValue(2);
}
```

① A plain list, with no synchronization. The realm guards it.

② A command runs in the context it was issued in, so what it changes can be traced to where it came from.

③ `submit` returns at once, and the command runs when the pump is drained.

A realm waypoint is a waypoint whose every descendant reaches the same guarded state. A command's closure captures the waypoint, so a command knows both where the execution is and what it may touch.

5.6. Telemetry

A telemetry waypoint adds a span to another waypoint. A step forward is a child span, and a join is a span under the first parent that links to the others.

Listing 19. The tree of waypoints is the tree of spans

```
try (PublishingTracerProvider provider = new PublishingTracerProvider(TracerProvider.noop())) {
①
    Tracer tracer = provider.get("orders");

    StateWaypoint<String, Integer> root = Waypoints.root("order", 0);
    TelemetryWaypoint<String> order = Waypoints.traced(root, tracer, element -> "step " +
element); ②
    TelemetryWaypoint<String> price = order.follow(root.commit("price", 120)); ③
    TelemetryWaypoint<String> stock = order.commit("stock");
    TelemetryWaypoint<String> ship = price.merge("ship", List.of(stock)); ④

    PublishingSpan span = (PublishingSpan) ship.getSpan();
    assertThat(span.getName()).isEqualTo("step ship");
    assertThat(span.getParent()).isSameAs(price.getSpan());
    assertThat(span.getParent().getParent()).isSameAs(order.getSpan());
    assertThat(ship.getParents()).isEqualTo(List.of(price, stock));

    assertThat(ship.facet(StateWaypoint.class).orElseThrow().getState()).isEqualTo(120); ⑤
    assertThat(ship.call(Span::current)).isSameAs(span); ⑥

    for (TelemetryWaypoint<String> waypoint: List.of(ship, stock, price, order)) {
        waypoint.close();
    }
    assertThat(span.isEnded()).isTrue();
}
```

① The runtime's publishing tracer: spans can be examined with no telemetry backend.

② Telemetry decorates a waypoint, here one that carries state.

③ A step that changed the state is followed: the caller advances, and telemetry adds the span.

④ A join. Its span's parent is the span of `price`, and it links to the span of `stock`.

⑤ What the decorated waypoint carries is reached through `facet`.

⑥ Work run through the waypoint sees its span as the current one.

Troubleshooting then happens at the level of spans and not of log lines. For any step there is, in one place, what it was, how long it took, what it cost, and the waypoints that led to it.

5.7. What changed on the way into the SDK

The original design returned Reactor's `Mono` from an asynchronous twin of every method, and split the work over seven modules. Four things are different here.

- **No reactive library.** On virtual threads a blocking call is cheap, which is also why `Flows` has no time operators. Methods are synchronous. Work that leaves the process, such as a call to a model provider, returns a `CompletionStage` tracked by the pump. That removed about half of the interface.
- **Telemetry is in the core.** The OpenTelemetry API is part of the foundation, and the pump already carries the context.
- **One way to add what a waypoint carries.** `facet(Class)`, as `capability(Class)` is for volumes. Composed interfaces multiply with every facet, so there are none beyond the three in the package.
- **A package in `common`,** not a module, since the pump should be a realm and bindings may carry waypoints. It becomes a module of its own if it grows.

5.8. Designed, not built

The rest of the original design carries over as intent.

History in Git

PLANNED A waypoint can be a real commit: the element and the state are written to a repository, and branches and merges happen in Git. That gives a durable history for long-running work, rollback as a reset, and a record for whoever examines it later.

Each such commit also records the trace context of the span that made it, as trailers in the W3C format, the same convention as `Signed-off-by`:

```
Waypoint-Id: 3f9a1c
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
```

Whoever reads the commit later, to resume or to audit, continues the same trace. A trace then spans not only processes and machines but time: an agent's work, a pause of days and its completion are one trace. It needs JGit, so it belongs with a Git volume, outside `common`.

One ladder from memory to a remote

PLANNED Where an agent keeps its state is a choice of root waypoint and not of code. The code that commits, merges and issues commands is the same on every rung.

Where the state is	What it buys
In memory	The fastest working state, lost when the process ends
A resource set with change recording	The same, with each step reversible in the process
A saved resource	Survives a crash
A commit in a local repository	A checkpoint that cannot change, with branch, merge and reset
A commit pushed to a remote	Survives the loss of the machine

With EMF the choice is per resource and not per agent: scratch work stays in memory while the resources that matter are committed. Recovery picks the rung it needs: reverse a change, reload a resource, reset to a commit, fetch from a remote.

A commit is an escalation

PLANNED When an agent reaches a point that needs a person, an approval or a choice between two readings, it commits a waypoint whose tree is the state of the model and whose message says who is asked, by when, how often to remind, and what to do if nobody answers. Then the process can end. A hook on the repository notifies and sets the timers. The answer arrives as another commit, made by a person or by the timeout, and the agent that resumes cannot tell which.

This is the pause and resume of workflow engines, built from what Git already has. The one part Git does not have is a scheduler, so the timers need a service outside it, and that service, not Git, knows whether a request has timed out.

A loop that builds within a budget

PLANNED The pieces compose into a loop that works unattended. The element is a thing to build, taken from a model of what is wanted. The state carries what is known and what budget remains. Each pass selects the most valuable thing that can be built now, checks that enough is known and enough budget is left, and either builds it, as one transaction that also spends the budget, or asks the person who can supply what is missing and moves on to the next thing that can be built. The spend is the difference between a waypoint's state and its parent's, so the ledger is auditable by construction. Requests to people are limited by the people, who say how often they will be asked: back pressure applied to human attention.

Mapping

PLANNED A mapping from one model to another is a traversal in which each step applies a rule to a source element. With a waypoint per step, "why does this target element exist" is answered by walking parents, and a rule that asks a model provider is recorded the same way as a rule that does not. Chapter 14 takes this up.

5.9. Open questions

- Whether a waypoint tracks its children. Here it does not, so that a long traversal does not hold all of its history in memory. A walker over a factory that records children is the alternative.
- Whether a waypoint has an identity of its own, or only where it is persisted. Here equality is identity, as it is for commits.
- Whether the realm's `adapt` needs a snapshot form beside the lens, for a view that is expensive to compute.

Chapter 6. Widgets

Capability	A view written once, against kinds such as text, link, section, list and table, and rendered into any format that provides them: Markdown, a terminal, and later AsciiDoc, HTML with Bootstrap and a diagram.
Demonstration	`RoadmapViewTests` in `tests/widgets`: one view of this roadmap's runtime items, rendered four ways. Its output is in this chapter.
Status	**SKETCH** in `runtime/widget`. Markdown, GitHub Flavored Markdown and a neutral terminal context are in the module. AsciiDoc, HTML and interaction are **PLANNED**.
Depends on	The runtime's `common` for the capability framework, and Chapter 4 for values that change.
Value and effort	High for medium. The polymorphic user interface has a vocabulary in the model and nothing at the level of Java. This is the missing half.
How to help	Write a context for a format you need. AsciiDoc is the next one, and its design is below.

6.1. Why

The runtime chooses a view by the class of what it shows. The SDK book's example of that returns an HTML node from every view, so every view is an HTML view, and a second format means a second set of views.

A view written against kinds does not name a format. It asks for a table and says what the columns and the rows are, and the format decides what a table is. The choice then has two halves, the class of the element and the format asked for, and this module is the second half.

The split is the one the `volume` module makes. Developers write against an API, and a

format implements an SPI, as `PathVolume` implements `Volume`. Widget toolkits have always been built this way, and the ones that failed did so by offering only what every platform had. So from the first version there are two ways out: a kind that a format does not have is simply absent, and a format that can do better than the neutral kind provides its own.

6.2. A widget

Listing 20. A widget renders itself into a context

```
@FunctionalInterface
public interface Widget<R, C> {

    R render(C context);

}
```

The context's type is the format: `MarkdownContext`, `TerminalContext`. What rendering gives back depends on the format too: nothing for Markdown, which is written and not built, and the number of rows used for a terminal.

A widget is asked for by the **type** of the context and given the **instance** only when it renders. The reasons are practical. A terminal draws many times a second, each time into a new area, so a widget bound to an instance would hold a stale one. Nested widgets render into derived contexts, a sub-area or a deeper heading level, and one child may render into many of them. And a widget built before any context exists is a value: it can be passed around and rendered into two terminals, or into a file and into an answer to an agent.

6.3. One view, four renderings

The demonstration is a view of this roadmap's own runtime items.

Listing 21. The data

```
enum Status {

    AVAILABLE(Variant.SUCCESS),
    SKETCH(Variant.WARNING),
    PLANNED(Variant.SECONDARY);
```

```
    final Variant variant;

    Status(Variant variant) {
        this.variant = variant;
    }

}

record Item(String name, String module, Status status) {

}

static final List<Item> ITEMS = List.of(
        new Item("Bindings", "runtime/common", Status.SKETCH),
        new Item("Waypoints", "runtime/common", Status.SKETCH),
        new Item("Widgets", "runtime/widget", Status.SKETCH),
        new Item("Terminal on TamboUI", "runtime/tui", Status.SKETCH),
        new Item("Nasdanika Commander", "runtime/commander", Status.SKETCH),
        new Item("Screens drawn in Draw.io", "none yet", Status.PLANNED));
```

Listing 22. The view names no format

```
static <C> Widget<?, C> roadmap(Kinds<C> kinds, Supplier<String> runtime, List<Item> items) {
    Text<C> text = kinds.require(Text.class);
    Link<C> link = kinds.require(Link.class);
    Badge<C> badge = kinds.require(Badge.class);
    Section<C> section = kinds.require(Section.class);
    ItemList<C> list = kinds.require(ItemList.class);
    Optional<Table<C>> table = kinds.find(Table.class);

    Widget<?, C> body = table
            .<Widget<?, C>>map(kind -> kind.table(
                    List.of(
                            new Table.Column<Item, C>(text.text("Item"), item -> text.text(item
.name()), 4),
                            new Table.Column<Item, C>(text.text("Module"), item -> text.text(
item.module()), 3),
                            new Table.Column<Item, C>(text.text("Status"), item -> badge.badge
(item.status().name(), item.status().variant), 2)),
                    Table.Rows.of(items)))
            .orElseGet(() -> list.list(false, items.stream().map(item -> text.text(item.name() +
", " + item.status())).toList()));

    return section.section(text.text("Runtime roadmap"), List.of(
            text.text(() -> "Built on runtime " + runtime.get() + "."),
            body,
            link.link(text.text("roadmap.sdk.nasdanika.org"),
"https://roadmap.sdk.nasdanika.org")));
}
```

`Kinds<C>` holds the kinds available for a context type. `require` gives a kind or fails, and `find`

gives an `Optional`, for a kind the view can do without. Here the view asks for a table and falls back to a list.

In GitHub Flavored Markdown there are tables:

```
StringBuilder out = new StringBuilder();
roadmap(Kinds.of(GfmContext.class), RUNTIME, ITEMS).render(new GfmContext(out));
```

Listing 23. What the test wrote

```
# Runtime roadmap

Built on runtime 2026.11.0.

| Item | Module | Status |
| --- | --- | --- |
| Bindings | runtime/common | **SKETCH** |
| Waypoints | runtime/common | **SKETCH** |
| Widgets | runtime/widget | **SKETCH** |
| Terminal on TamboUI | runtime/tui | **SKETCH** |
| Nasdanika Commander | runtime/commander | **SKETCH** |
| Screens drawn in Draw.io | none yet | **PLANNED** |

[roadmap.sdk.nasdanika.org](<https://roadmap.sdk.nasdanika.org>)
```

CommonMark has no tables, so no table kind is provided for `MarkdownContext`, and the same view renders a list:

Listing 24. The same view in CommonMark

```
# Runtime roadmap

Built on runtime 2026.11.0.

- Bindings, SKETCH
- Waypoints, SKETCH
- Widgets, SKETCH
- Terminal on TamboUI, SKETCH
- Nasdanika Commander, SKETCH
- Screens drawn in Draw.io, PLANNED

[roadmap.sdk.nasdanika.org](<https://roadmap.sdk.nasdanika.org>)
```

On a terminal, as a grid of characters with no terminal library involved:

```
GridContext screen = new GridContext(64, 12);
```

```
roadmap(Kinds.of(TerminalContext.class), RUNTIME, ITEMS).render(screen);
```

Listing 25. The same view as text

```
Runtime roadmap
  Built on runtime 2026.11.0.
  Item                      Module               Status
  ------------------------- -------------------- --------------
  Bindings                  runtime/common       SKETCH
  Waypoints                 runtime/common       SKETCH
  Widgets                   runtime/widget       SKETCH
  Terminal on TamboUI       runtime/tui          SKETCH
  Nasdanika Commander       runtime/commander    SKETCH
  Screens drawn in Draw.io  none yet             PLANNED
  roadmap.sdk.nasdanika.org
```

And on a TamboUI screen, where a section is a box and a status has a color. Chapter 7 says how.

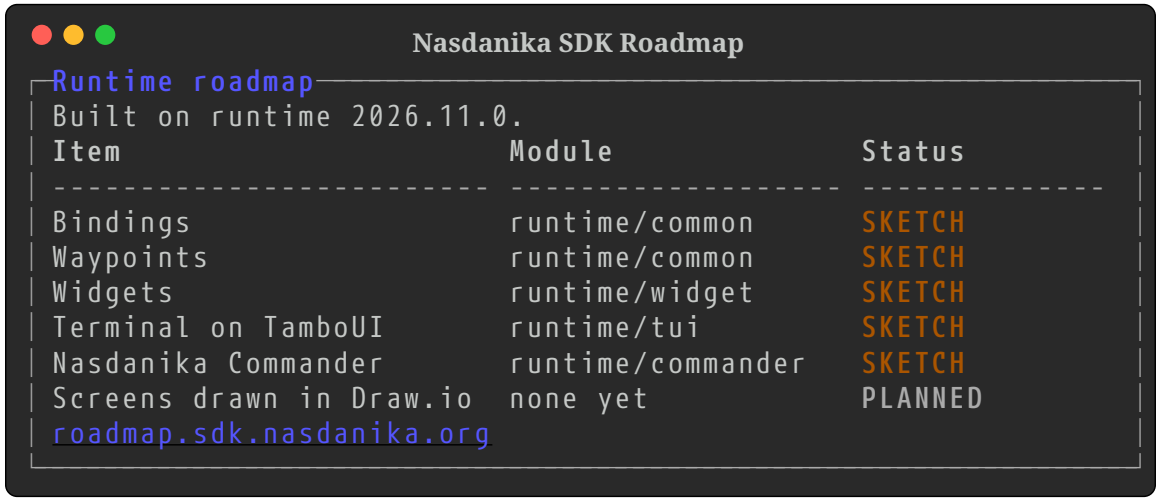


Figure 1. The same view on a TamboUI screen, as the test drew it

The table of items above is therefore not written by hand anywhere. It is this roadmap rendering a part of itself with the module it describes, and Appendix A may be generated the same way once the items are a model.

6.4. How a kind is found

A kind is resolved through the capability framework. The requirement is an explicit record:

```
public record KindOf(Class<?> kind, Class<?> contextType) {

    public KindOf {
        Objects.requireNonNull(kind, "kind");
        Objects.requireNonNull(contextType, "contextType");
    }

}
```

A format contributes a factory with the kinds it provides, each for a context type:

Listing 26. The kinds of Markdown

```
public class MarkdownKinds extends KindFactory {

    public MarkdownKinds() {
        super(
            new KindProvider<>(Text.class, MarkdownContext.class, new MarkdownText()),
            new KindProvider<>(Link.class, MarkdownContext.class, new MarkdownLink()),
            new KindProvider<>(Section.class, MarkdownContext.class, new MarkdownSection()),
            new KindProvider<>(ItemList.class, MarkdownContext.class, new MarkdownItemList()),
            new KindProvider<>(Badge.class, MarkdownContext.class, new MarkdownBadge()),
            new KindProvider<>(Table.class, GfmContext.class, new GfmTable())); ①
    }
```

① The table is for `GfmContext` only. Everything else serves both, since a `GfmContext` is a `MarkdownContext`.

The factory is registered as a service in the module's descriptor, as every capability factory is:

```
provides CapabilityFactory with
    MarkdownKinds,
    TerminalKinds;
```

Resolution follows the context's class hierarchy, as views follow the supertypes of a class:

Listing 27. Along the hierarchy

```
Kinds<MarkdownContext> markdown = Kinds.of(MarkdownContext.class);
```

```
Kinds<GfmContext> gfm = Kinds.of(GfmContext.class);

Text<MarkdownContext> markdownText = markdown.require(Text.class);
Text<GfmContext> gfmText = gfm.require(Text.class);
assertThat((Object) gfmText).isSameAs(markdownText); ①

assertThat(gfm.find(Table.class)).isPresent();
assertThat(markdown.find(Table.class)).isEmpty(); ②
assertThatThrownBy(() -> markdown.require(Table.class))
        .isInstanceOf(UnsupportedOperationException.class)
        .hasMessage("No Table for " + MarkdownContext.class.getName());
```

① No text kind is written for `GfmContext`. The one for `MarkdownContext` answers.

② A missing kind means that the format does not have the feature, and the view is told so.

Listing 28. The most specific provider wins

```
@SuppressWarnings("unchecked")
public <K> Optional<K> find(Class<? super K> kind) {
    return (Optional<K>) resolved.computeIfAbsent(kind, this::resolve);
}

private Optional<Object> resolve(Class<?> kind) {
    List<KindProvider<?>> providers = loader.loadAll(new KindOf(kind, contextType)); ①
    KindProvider<?> best = null;
    for (KindProvider<?> provider: providers) {
        if (best == null || isMoreSpecific(provider, best)) { ②
            best = provider;
        } else if (provider.contextType() == best.contextType() && provider.implementation() !=
best.implementation()) {
            Telemetry.log( ③
                    Telemetry.logger(Telemetry.current()),
                    Severity.WARN,
                    "widget.kind.ambiguous",
                    "More than one " + kind.getSimpleName() + " for " + best.contextType
().getName() + ", the first one is used",
                    Attributes.empty());
        }
    }
    return Optional.ofNullable(best).map(KindProvider::implementation);
}

private static boolean isMoreSpecific(KindProvider<?> candidate, KindProvider<?> current) {
    return candidate.contextType() != current.contextType() && current.contextType
().isAssignableFrom(candidate.contextType());
}
```

① Every provider whose context type is the one asked for, or a supertype of it.

② A provider for a subtype is more specific than one for its supertype.

③ Two providers for the same context type are a tie, and a tie is reported.

Listing 29. A format that can do better says so

```
static final Text<GfmContext> SHOUTING = (value, format) -> context -> {
    context.emit(context.escape(value.get().toUpperCase()));
    return null;
};

static final class ShoutingKinds extends KindFactory {

    ShoutingKinds() {
        super(new KindProvider<>(Text.class, GfmContext.class, SHOUTING));
    }

}

@Test
void aKindForAMoreSpecificContextWins() {
    CapabilityLoader loader = CapabilityLoader.of(new MarkdownKinds(), new ShoutingKinds()); ①

    Text<GfmContext> gfmText = Kinds.of(GfmContext.class, loader).require(Text.class);
    Text<MarkdownContext> markdownText = Kinds.of(MarkdownContext.class, loader).require(Text
.class);

    assertThat((Object) gfmText).isSameAs(SHOUTING); ②
    assertThat((Object) markdownText).isNotSameAs(SHOUTING);
}
```

① A loader with Markdown's kinds and one more factory.

② For `GfmContext` the specific kind is found, and for `MarkdownContext` the neutral one still is.

Resolutions are remembered per kind and context type. Each one is traced, like every capability resolution, so the trace of a rendering says which kinds were used and where they came from.

6.5. Text carries its format

A plain string is never read as markup.

Listing 30. A value cannot inject markup

```
Text<MarkdownContext> text = Kinds.of(MarkdownContext.class).require(Text.class); ①
Widget<?, MarkdownContext> widget = text.text("<script>alert(1)</script> *bold*
[x](javascript:alert(1))"); ②

StringBuilder out = new StringBuilder();
```

```
widget.render(new MarkdownContext(out)); ③

assertThat(out.toString()).isEqualTo("\\<script\\>alert(1)\\</script\\> \\*bold\\*
\\[x\\](javascript:alert(1))\n\n");
```

① The text kind for Markdown.

② A value that looks like HTML, emphasis and a link with a script in it.

③ Rendered, it is that text and nothing else.

That is a security boundary and not only a convenience. What a view shows comes from models, from files and from people, and none of it should be able to add a link, a script or a heading to the output. The same rule holds on a terminal, where the danger is an escape sequence:

```
Text<TerminalContext> text = Kinds.of(TerminalContext.class).require(Text.class);
GridContext screen = new GridContext(30, 1);

text.text("x\u001b]8;;http://evil\u0007link").render(screen); ①

assertThat(screen.toString()).isEqualTo("x]8;;http://evillink\n");
```

① The bytes that would start a hyperlink in a terminal. Control characters never reach the screen.

Text that **is** markup says so. Text in the target's own format is passed through, and text in another format needs a conversion:

```
Text<MarkdownContext> text = Kinds.of(MarkdownContext.class).require(Text.class);
List<String> diagnostics = new ArrayList<>();
StringBuilder out = new StringBuilder();
MarkdownContext context = new MarkdownContext(out, 1, MarkdownContext.Mode.BLOCK, (severity,
message) -> diagnostics.add(severity + ": " + message));

text.text(() -> "Balance is **overdue**", Text.MARKDOWN).render(context); ①
text.text(() -> "Balance is <b>overdue</b>", Text.HTML).render(context); ②

assertThat(out.toString()).isEqualTo("Balance is **overdue**\n\nBalance is
\\<b\\>overdue\\</b\\>\n\n");
assertThat(diagnostics).containsExactly("WARN: No conversion from html to Markdown, rendered as
plain text");
```

① Markdown into Markdown: passed through.

② HTML into Markdown: no conversion exists yet, so it is reported and rendered as plain

text, which is the safe direction.

Diagnostics go through the context and not into what a widget returns. By default a diagnostic is a log record in the current span.

Conversions between formats are PLANNED to go through the document model, as Pandoc's go through its own: a reader and a writer per format, not a converter per pair.

6.6. Markdown

A Markdown context knows how deep it is in sections and whether it is writing a block, text within a line, or a table cell. Nested widgets render into derived contexts. A section renders its body one heading level down:

```
private static final class MarkdownSection implements Section<MarkdownContext> {

    @Override
    public Widget<Void, MarkdownContext> section(Widget<?, ? super MarkdownContext> title, List<?
extends Widget<?, ? super MarkdownContext>> body) {
        return context -> {
            if (context.getMode() != Mode.BLOCK) {
                context.diagnostic(Severity.WARN, "A section inside a line or a table cell: only
its title is rendered");
                context.emit(render(title, context, context.getMode()));
                return null;
            }
            context.emit("#".repeat(Math.min(context.getLevel(), 6)) + " " + render(title,
context, Mode.INLINE));
            MarkdownContext nested = context.nested(); ①
            for (Widget<?, ? super MarkdownContext> widget: body) {
                widget.render(nested);
            }
            return null;
        };
    }

}
```

① The body is rendered into a derived context, one level deeper. The widgets in it do not know their level.

A list item, a link's label and a table cell render into a buffer of their own, which the enclosing widget places:

```
public MarkdownContext capture(Mode mode) {
    return derive(new StringBuilder(), level, mode, diagnostics);
}

/**
 * @return What was written into a context made by {@link #capture(Mode)}, without the blank
lines at its end
 */
public String captured() {
    if (out instanceof StringBuilder buffer) {
        return buffer.toString().stripTrailing();
    }
    throw new IllegalStateException("Not a capturing context");
}
```

Escaping depends on where the text goes:

```
public String escape(String plain) {
    StringBuilder escaped = new StringBuilder(plain.length() + 16);
    boolean lineStart = true;
    for (int i = 0; i < plain.length(); ++i) {
        char ch = plain.charAt(i);
        switch (ch) {
            case '\\', '`', '*', '_', '[', ']', '<', '>', '~' -> escaped.append('\\').append(ch);
①
            case '&' -> escaped.append("&amp;");
            case '|' -> escaped.append(mode == Mode.CELL ? "\\|" : "|"); ②
            case '#', '+', '-', '=' -> escaped.append(lineStart ? "\\" : "").append(ch); ③
            case '.', ')' -> escaped.append(lineStart(plain, i) ? "\\" : "").append(ch); ④
            case '\r' -> { }
            case '\n' -> escaped.append(mode == Mode.BLOCK ? "\n" : " "); ⑤
            default -> escaped.append(Character.isISOControl(ch) ? ' ' : ch);
        }
        lineStart = ch == '\n' || (lineStart && ch == ' ');
    }
    return escaped.toString();
}
```

① Characters that are markup anywhere.

② A pipe ends a cell only in a table.

③ Characters that start a block only at the start of a line.

④ A dot or a parenthesis after digits at the start of a line would start a numbered list.

⑤ Within a line or a cell, a line break is a space.

6.7. Surfaces and the terminal

Some contexts are areas: a rectangle that widgets draw into and divide among the widgets inside them.

```
public interface SurfaceContext extends RenderContext {

    double getWidth();

    double getHeight();

    /**
     * A context for a part of this area, with coordinates relative to this area. The part is
clipped
     * to this area: a widget cannot draw outside what it was given.
     */
    SurfaceContext area(double x, double y, double width, double height);

}
```

The terminal is one such surface, with cells as its unit. A Draw.io page is another, with pixels, and Chapter 9 comes back to it.

```
public interface TerminalContext extends SurfaceContext {

    @Override
    TerminalContext area(double x, double y, double width, double height);

    default TerminalContext area(int column, int row, int columns, int rows) {
        return area((double) column, (double) row, (double) columns, (double) rows);
    }

    default int getColumns() {
        return (int) getWidth();
    }

    default int getRows() {
        return (int) getHeight();
    }

    /**
     * Writes text on one row, starting at a column, both relative to this area. What does not
fit the
     * area is not written, and control characters never reach the terminal.
     */
    void put(int column, int row, String text, TextStyle style);

    /**
     * A context for the same area in which text takes the given style unless it brings its own:
```

```
how
     * a section makes its title bold without knowing what the title is.
     */
    TerminalContext styled(TextStyle style);

}
```

`TerminalContext` abstracts only what the neutral kinds need: an area, text at a position, and a style in logical terms. Colors are logical too, named for what they mean: primary, success, danger. The names are Bootstrap's, so a view says "danger" once, and a page, a terminal's palette and a diagram each show it their own way.

A terminal widget gives back the rows it used, which is how widgets are stacked without a pass to measure them first:

```
private static final class TerminalSection implements Section<TerminalContext> {

    private static final int INDENT = 2;

    @Override
    public Widget<Integer, TerminalContext> section(Widget<?, ? super TerminalContext> title,
List<? extends Widget<?, ? super TerminalContext>> body) {
        return context -> {
            int row = rows(title, context.styled(TextStyle.PLAIN.withBold().withVariant(Variant
.PRIMARY))); ①
            for (Widget<?, ? super TerminalContext> widget: body) {
                TerminalContext area = context.area(INDENT, row, context.getColumns() - INDENT,
context.getRows() - row); ②
                row += rows(widget, area); ③
            }
            return row;
        };
    }

}
```

① The title, in a context where text is bold and in the primary color unless it says otherwise.

② Each widget of the body gets what is left of the area, indented.

③ And says how many rows it used.

A widget cannot draw outside the area it was given, and a table reads only the rows that fit. A table of a million rows on a screen of five reads three of them.

`GridContext` implements the terminal context over a grid of characters in memory. It is the smallest provider, and a useful one: a test asserts on it, a log can include it, and an agent can read a screen as text.

6.8. A binding is a value

Every value a kind takes is a supplier, so a binding from Chapter 4 is a value.

```
ValueBinding<String> asOf = Bindings.value("2026-11-11");
Widget<?, TerminalContext> view = accounts(Kinds.of(TerminalContext.class), asOf, ACCOUNTS); ①

GridContext first = new GridContext(44, 3);
view.render(first);
asOf.set("2026-11-12");
GridContext second = new GridContext(44, 3);
view.render(second); ②

assertThat(first.toString()).contains("Balances as of 2026-11-11.");
assertThat(second.toString()).contains("Balances as of 2026-11-12.");
```

① The date is a binding. The view does not know.

② The same widget, rendered again, reads the value as it is now.

A static format reads the value when it renders, and a screen reads it on every frame. The widget is the same.

State that outlives one rendering, a scroll position or a selection, is not the widget's. The screen owns it and passes it in, as TamboUI and Ratatui do, or the widget reads it through a binding. Otherwise binding late to a context would only move the early binding into the widget's own fields.

6.9. What is not there yet

- PLANNED **An AsciiDoc context.** AsciiDoc is a format of several files. Widgets ask for sections, and the context decides through a policy which sections become files of their own: by level, as a book splits into chapters, or by class, as a reference has a file per class. A locator maps a model element to a path and an anchor, so a link to an element is written before the element is rendered. `Include` is a kind that only AsciiDoc has.

- PLANNED **The first consumers:** the glossary of a Meta package, rendered as Markdown and AsciiDoc with the bank model as the example, and the documentation of a command line, rendered from its saved command tree (Chapter 12).
- PLANNED **An HTML context with Bootstrap,** beside the HTML models it depends on, and a context that builds the document model of Chapter 11.
- PLANNED **Interaction:** `Action`, `Selection` and `Input` as kinds that static formats do not provide, and `Tree` and a form, each arriving with the first screen that needs it.
- PLANNED **A table that scrolls,** with its position in a state object.

6.10. Why not something that exists

The question will be asked, and the survey was made. Nothing found covers documents and screens with one vocabulary in Java.

- **Maven Doxia** has a sink with calls for document elements, and writers for several formats. It is for documents only, and its list of calls is a good checklist for the kinds.
- **json-render and Adaptive Cards** are catalogs of components rendered by several hosts. They are the closest in idea and are not Java libraries. Each could be a context here.
- **JSON Forms and EMF Forms** render forms from a model and write no documents.
- **Pandoc** converts between document formats through one model, which is the plan for conversions here, and has no screens.

The answer to "why not" is then the same each time: this is a small API over what a format already has, and where a format has a library, the library is its provider.

Chapter 7. The Terminal

Capability	User interfaces in a terminal for people who live in one: small screens for a task, launched by a command, on the same kinds as documents and pages. And screenshots of those screens as figures, taken by a test.
Demonstration	`TamboUITests` in `runtime/tui`, the last rendering of Chapter 6, and Chapter 8, which is built on it.
Status	**SKETCH** in `runtime/tui`. The base terminal command, the kit and telemetry for commands are **PLANNED**.
Depends on	Chapter 6, and TamboUI 0.7.0.
Value and effort	High for medium. For technical people a terminal is the first user interface to offer, and it needs no server and no browser.
How to help	Use it in a terminal you care about and report what the keys and the colors do there. Try a screen of your own.

7.1. Why a terminal

A terminal is where developers, operators and architects already are. A screen there starts in a moment, works over SSH, and runs in the terminal of an IDE. It is also the natural form of the small user interfaces the first chapter spoke of: one screen for one task, launched by a command, not a portal.

The order follows from that. For technical people the terminal comes first and the browser second, and React remains the way to reach people who are not technical.

TamboUI is the library. It is a Java port of the ideas of Ratatui: a buffer of cells, widgets that render into an area of it, and state kept apart from the widgets. It is at version 0.7.0, so the rule from Chapter 6 matters here: neutral widgets are written against `TerminalContext` and never import TamboUI, and what changes in TamboUI changes in one module.

7.2. TamboUI as a provider

`TamboUIContext` is a terminal context over an area of a TamboUI buffer.

```
public class TamboUIContext implements TerminalContext {

    private final Buffer buffer;
    private final Rect rect;
    private final TextStyle base;

    /**
     * A context for a whole buffer.
     */
    public TamboUIContext(Buffer buffer) {
        this(buffer, buffer.area(), null);
    }

    /**
     * A context for an area of a buffer, as TamboUI passes them to a widget.
     */
    public TamboUIContext(Buffer buffer, Rect rect) {
        this(buffer, rect, null);
    }

    private TamboUIContext(Buffer buffer, Rect rect, TextStyle base) {
        this.buffer = Objects.requireNonNull(buffer, "buffer");
        this.rect = Objects.requireNonNull(rect, "rect");
        this.base = base;
    }

    /** The buffer, for widgets written for TamboUI. */
    public Buffer getBuffer() {
        return buffer;
    }

    /** The area of the buffer this context draws into, in the buffer's coordinates. */
    public Rect getRect() {
        return rect;
    }
    // ...
}
```

Writing text is clipped to the area and cleaned of control characters, as the grid of Chapter 6 does:

```
@Override
public void put(int column, int row, String text, TextStyle style) {
    if (row < 0 || row >= rect.height() || column >= rect.width()) {
        return;
    }
```

```
    String clean = GridContext.sanitize(text); ①
    int from = Math.max(0, -column);
    int to = Math.min(clean.length(), rect.width() - column); ②
    if (from < to) {
        buffer.setString(rect.x() + column + from, rect.y() + row, clean.substring(from, to),
style(style.over(base)));
    }
}
```

① The same cleaning as every provider: a value cannot send an escape sequence.

② What does not fit the area is not written.

Logical styles become the terminal's own:

```
public static Style style(TextStyle style) {
    Style result = Style.EMPTY;
    if (style.variant() != null) {
        result = result.fg(color(style.variant()));
    }
    if (style.bold()) {
        result = result.bold();
    }
    if (style.underlined()) {
        result = result.underlined();
    }
    if (style.reversed()) {
        result = result.reversed();
    }
    if (style.link() != null && style.link().chars().noneMatch(Character::isISOControl)) { ①
        result = result.hyperlink(style.link());
    }
    return result;
}

public static Color color(Variant variant) {
    return switch (variant) {
        case PRIMARY -> Color.LIGHT_BLUE; // Plain blue is too dark to read on a dark background
        case SECONDARY -> Color.GRAY;
        case SUCCESS -> Color.GREEN;
        case DANGER -> Color.RED;
        case WARNING -> Color.YELLOW;
        case INFO -> Color.CYAN;
        case LIGHT -> Color.WHITE;
        case DARK -> Color.DARK_GRAY;
    };
}
```

① A link becomes a hyperlink of the terminal, unless its target has control characters in it.

A view can name only logical colors, and they map to the sixteen colors every terminal

has. So a screen follows the palette the user chose for their terminal, light or dark, instead of bringing its own.

7.3. The way out, in use

`TamboUIContext` extends `TerminalContext`, so every neutral kind serves it. And it hands out the buffer and the area, so a kind written for it can use whatever TamboUI has.

Listing 31. Neutral kinds serve TamboUI, and a specific one wins

```
Kinds<TamboUIContext> tamboui = Kinds.of(TamboUIContext.class);
Kinds<TerminalContext> terminal = Kinds.of(TerminalContext.class);

Text<TamboUIContext> text = tamboui.require(Text.class);
Text<TerminalContext> neutralText = terminal.require(Text.class);
assertThat((Object) text).isSameAs(neutralText); ①

Section<TamboUIContext> section = tamboui.require(Section.class);
Section<TerminalContext> neutralSection = terminal.require(Section.class);
assertThat((Object) section).isNotSameAs(neutralSection); ②
```

① No text kind is written for TamboUI. The neutral one answers.

② A section is written for TamboUI, and it is found before the neutral one.

The neutral section indents its body. The TamboUI section draws a block with a border and the title on it. The neutral kinds draw with ASCII only, so that their output is good as plain text anywhere, and a border is what this context adds:

```
private static final class BlockSection implements Section<TamboUIContext> {

    @Override
    public Widget<Integer, TamboUIContext> section(Widget<?, ? super TamboUIContext> title,
List<? extends Widget<?, ? super TamboUIContext>> body) {
        return context -> {
            Rect rect = context.getRect();
            if (rect.width() < 6 || rect.height() < 3) {
                return TerminalKinds.rows(title, context);
            }

            // The body, drawn off screen first: its height is the height of the box
            Buffer scratch = Buffer.empty(Rect.of(rect.width() - 4, rect.height() - 2));
            TamboUIContext inner = new TamboUIContext(scratch); ①
            int rows = 0;
            for (Widget<?, ? super TamboUIContext> widget: body) {
                rows += TerminalKinds.rows(widget, inner.area(0, rows, inner.getColumns(), inner
```

```
.getRows() - rows));
            }
            rows = Math.min(rows, rect.height() - 2);

            Block.builder() ②
                    .borders(Borders.ALL)
                    .borderType(BorderType.PLAIN)
                    .borderColor(TamboUIContext.color(Variant.SECONDARY))
                    .build()
                    .render(new Rect(rect.x(), rect.y(), rect.width(), rows + 2), context
.getBuffer());

            for (int y = 0; y < rows; ++y) {
                for (int x = 0; x < scratch.width(); ++x) {
                    Cell cell = scratch.get(x, y);
                    if (!cell.isEmpty()) {
                        context.getBuffer().set(rect.x() + 2 + x, rect.y() + 1 + y, cell);
                    }
                }
            }

            TamboUIContext heading = context.styled(TextStyle.PLAIN.withBold().withVariant
(Variant.PRIMARY)).area(2, 0, rect.width() - 4, 1);
            TerminalKinds.rows(title, heading); ③
            return rows + 2;
        };
    }

}
```

① The body is drawn off screen first, since its height is the height of the box.

② TamboUI's own block widget draws the border.

③ The title is a widget like any other, rendered over the top border in a bold, primary style.

```
Kinds<TamboUIContext> kinds = Kinds.of(TamboUIContext.class);
Text<TamboUIContext> text = kinds.require(Text.class);
Section<TamboUIContext> section = kinds.require(Section.class);
ItemList<TamboUIContext> list = kinds.require(ItemList.class);

Buffer buffer = Screenshots.draw(30, 6, context -> section.section(
        text.text("Accounts"),
        List.of(list.list(false, List.of(text.text("Checking"), text.text("Savings"))))).render
(context));
```

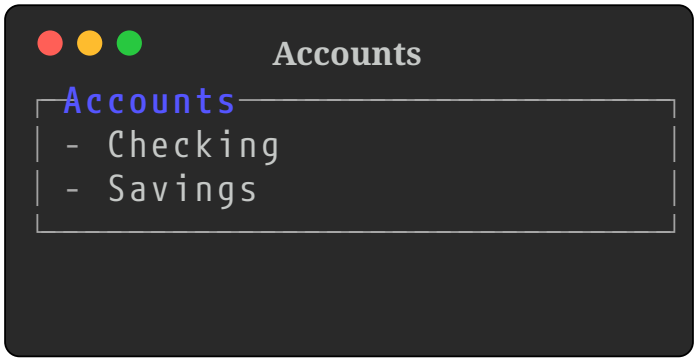


Figure 2. A section on a TamboUI screen

The view did not change. The same `roadmap` view of Chapter 6 is indented text in a grid and a box on a TamboUI screen, because it asked for a section and each context answered in its own way.

7.4. Screenshots

A screen draws into a buffer, and a buffer can be exported. So a screenshot needs no terminal: a test draws the screen into a buffer and writes it out.

```
Buffer buffer = Screenshots.draw(30, 3, context -> context.put(1, 1, "Balance  1,200.00",
TextStyle.PLAIN.withBold()));

String svg = Screenshots.svg(buffer, "Accounts"); ①
String selfContained = Screenshots.selfContained(svg); ②

assertThat(svg).startsWith("<svg").contains("Accounts").contains("@font-face");
assertThat(selfContained).startsWith("<svg").doesNotContain("@font-face").doesNotContain
("https://cdnjs");
assertThat(Screenshots.svg(buffer, "Accounts")).isEqualTo(svg); ③
```

① TamboUI's exporter writes the buffer as SVG, in a window frame with a title.

② The same SVG without its font rules, so that it loads nothing from the network.

③ The same screen gives the same SVG, byte for byte, so a figure in a repository changes only when the screen does.

The figures of this book are made that way. The last rendering in Chapter 6 is written by these lines of `RoadmapViewTests`:

```
Widget<?, TamboUIContext> view = roadmap(Kinds.of(TamboUIContext.class), RUNTIME, ITEMS); ①
```

```
Buffer buffer = Screenshots.draw(64, 12, view::render); ②

Path directory = Path.of(System.getProperty("roadmap.target", "target"), "showcase");
Screenshots.write(buffer, "Nasdanika SDK Roadmap", directory.resolve("roadmap.svg")); ③
```

① The view, for TamboUI.

② Drawn into a buffer of 64 by 12 cells.

③ Written as a self-contained SVG, which the book's build copies beside the manuscript.

Documentation of a screen then cannot drift from the screen: the picture is taken when the book is built.

Three things were learned about TamboUI 0.7.0 on the way, and they are candidates for a field note.

- **The exporters are in the core jar and not in its module's exports.** On the module path, the packages under `dev.tamboui.export` have to be exported to the module that calls them, with `--add-exports`. The `tui` module's build does that, and so does every module that takes screenshots. On the class path nothing is needed.
- **The SVG loads a font from a CDN,** Fira Code, and falls back to `monospace`. Each run of characters has its width set, so the grid holds in any monospace font. For a book, or behind a proxy, `Screenshots.selfContained` removes the font rules.
- **The text exporter uses the line separator of the platform.** `Screenshots.text` makes it a line feed everywhere, so an assertion holds on every machine.

7.5. What is not there yet

- PLANNED **A base terminal command.** A command that draws takes a terminal, runs a loop and restores the terminal when it ends, and that belongs in one place. It arrives with the second command that draws. `cli` stays free of TamboUI.
- PLANNED **The kit:** a form with fields and problems shown beside them, a status line that every command on a line can contribute to, and a way for a test to drive a screen. It is extracted from the commander and the first task, not designed ahead of them.
- PLANNED **Telemetry for commands:** the spans of a running command as a tree that updates, over the publishing tracer. It starts on JLine alone, with no new dependency, and reaches every command.

- PLANNED **Task screens.** The first candidates are small: asserting a balance, approving a step of a workflow.
- PLANNED **An SVG context,** beside the SVG model, for figures that are drawn and not captured.

The open question is the keys. The terminal inside an IDE keeps some function keys for itself, so every command needs a letter as well, and Chapter 8 gives each of its commands one.

Chapter 8. Nasdanika Commander

Capability	Two panels over volumes with the classic keys: navigate, view, edit, copy, move, create and delete. Over the local file system it is a file manager for wherever the usual one is not installed. Over other volumes it is a way into whatever they hold.
Demonstration	`CommanderTests` and `TerminalLoopTests` in `runtime/commander`. After the build, `java -jar` runs it.
Status	**SKETCH** in `runtime/commander`: a working prototype with the minimum for daily file work.
Depends on	Chapter 7, the runtime's `volume` module, TamboUI 0.7.0 and JLine.
Value and effort	High for medium. It is used by its author first, and it is the first real consumer of the terminal modules.
How to help	Use it for a day and report what you reached for and did not find. A volume of your own is the most interesting contribution: see Section 8.6.

8.1. Why

Anyone who used Norton Commander or FAR knows the screen: two panels, a bar of function keys, and hands that do not leave the keyboard. That knowledge is decades old and still in the fingers, so the keys are kept as they were.

The reason to build one more is what the panels show. A panel here is a `Volume`, the runtime's small file abstraction, and a directory on disk is only one kind of volume. A zip file, a repository on a server, the modules of a running JVM and the elements of a model can all be volumes. Wherever a volume exists, the commander is already its user interface, with no page to write and no server to run.

8.2. Running it

```
mvn -q package
java -jar runtime/commander/target/commander-2026.11.0.jar [left directory] [right directory]
```

Both panels start in the current directory unless told otherwise. The jar's manifest names its dependencies in `target/lib`, which the build fills, and `commander` and `commander.cmd` beside the module's `pom.xml` run the same line.

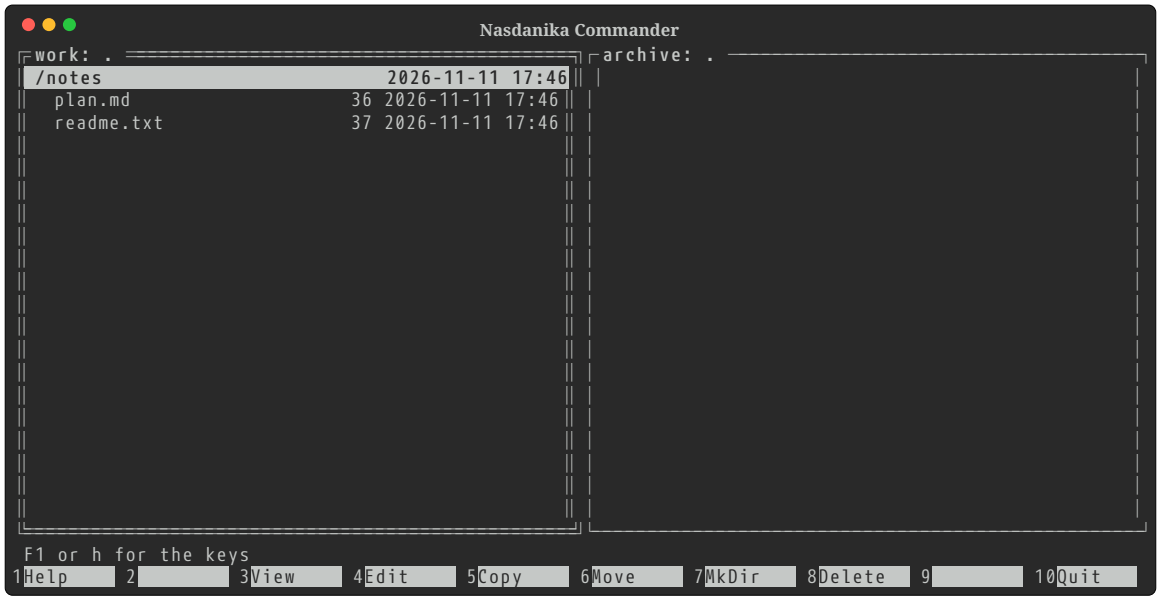


Figure 3. Two panels over two directories

Key	Letter	What it does
`Tab`		The other panel
`Enter`		Open a directory, view a file
`Backspace`		Up one directory
`F1`	h	The keys
`F3`	v	View a file
`F4`	e	Edit a file. `F2` or `Ctrl` + `S` saves, `Esc` closes
`Shift` + `F4`	n	Create a file and edit it
`F5`	c	Copy to the other panel

Key	Letter	What it does
F6	m	Move or rename
F7	k	Create a directory
F8	d	Delete
Ctrl + R	r	Read the panels again
F10	q	Quit

Every command has a letter, for terminals that keep function keys for themselves.

8.3. The application is not the terminal

`Commander` takes key events and draws frames, and knows nothing about where either comes from.

```
public void handle(KeyEvent key) {
    if (key.isRelease()) {
        return;
    }
    try {
        if (dialog != null) {
            dialog(key);
        } else {
            switch (mode) {
                case BROWSE -> browse(key);
                case VIEW -> view(key);
                case EDIT -> edit(key);
            }
        }
    } catch (IOException | RuntimeException e) {
        status = "Error: " + e.getClass().getSimpleName() + (e.getMessage() == null ? "" : ": " +
e.getMessage()); ①
    }
}
```

① A failure, such as a file that cannot be read, becomes the status line. It never ends the application.

```
public static void main(String[] args) throws Exception {
    Volume files = MountVolume.ofLocalFileSystem(); ①
    String leftDirectory = MountVolume.localPath(Path.of(args.length > 0 ? args[0] : ""));
    String rightDirectory = MountVolume.localPath(Path.of(args.length > 1 ? args[1] : args.length
> 0 ? args[0] : ""));
```

```
    Commander commander = new Commander(new Panel("files", files, leftDirectory), new Panel
("files", files, rightDirectory));

    run(commander, TuiConfig.builder().backend(new JLineBackend()).build()); ②
}

/**
 * Runs a commander in a terminal until it quits.
 */
public static void run(Commander commander, TuiConfig config) throws Exception {
    try (TuiRunner runner = TuiRunner.create(config)) {
        runner.run((event, tui) -> {
            if (event instanceof KeyEvent key) {
                commander.handle(key); ③
                if (!commander.isRunning()) {
                    tui.quit();
                }
                return true;
            }
            return false;
        }, commander::render);
    }
}
```

① The local file system as one volume: its root where there is one, and its drives mounted by name where there are several.

② The terminal backend is named, so the same line works on the class path and on the module path.

③ The loop does one thing: hand keys to the commander and let it draw.

Because of that split, a test is the application's second terminal. It presses keys and reads the screen from a buffer:

```
@BeforeEach
void setUp() throws IOException {
    work = Files.createDirectory(directory.resolve("work"));
    archive = Files.createDirectory(directory.resolve("archive"));
    Files.createDirectory(work.resolve("notes"));
    Files.writeString(work.resolve("notes").resolve("todo.txt"), "call the bank\n");
    Files.writeString(work.resolve("plan.md"), "# Plan\r\n\r\n1. Draw\r\n2. Run\r\n3. Bind\r\n");
①
    Files.writeString(work.resolve("readme.txt"), "A directory to try the commander on.\n");

    commander = new Commander(
            new Panel("work", Volume.of(work), "."), ②
            new Panel("archive", Volume.of(archive), "."));
}
```

① A file with Windows line endings, to see that editing keeps them.

② Each panel is a volume over a directory.

```
void press(KeyCode... keys) {
    for (KeyCode key: keys) {
        commander.handle(KeyEvent.ofKey(key));
    }
}

void type(String text) {
    text.chars().forEach(ch -> commander.handle(KeyEvent.ofChar(ch)));
}

/** What is on the screen, as text. */
String screen(int columns, int rows) {
    Buffer buffer = Screenshots.buffer(columns, rows);
    commander.render(Frame.forTesting(buffer)); ①
    return Screenshots.text(buffer);
}

/** A screenshot for the book, in target/screenshots. */
void screenshot(String name) {
    Buffer buffer = Screenshots.buffer(100, 24);
    commander.render(Frame.forTesting(buffer));
    Path file = Path.of(System.getProperty("roadmap.target", "target"), "screenshots", name +
".svg");
    Screenshots.write(buffer, "Nasdanika Commander", file); ②
    try {
        Files.writeString(file.resolveSibling(name + ".txt"), Screenshots.text(buffer));
    } catch (IOException e) {
        throw new java.io.UncheckedIOException(e);
    }
}
```

① A frame over a buffer, with no terminal.

② The screenshots of this chapter.

8.4. Viewing and editing

```
press(KeyCode.DOWN, KeyCode.F3); ①

assertThat(commander.getMode()).isEqualTo(Mode.VIEW);
assertThat(screen(100, 24)).contains(" View plan.md ").contains("# Plan").contains("3. Bind");

type("x"); ②
press(KeyCode.ESCAPE);
assertThat(commander.getMode()).isEqualTo(Mode.BROWSE);
```

① Down to `plan.md`, and view it.

② Typing does nothing in the viewer.

```
press(KeyCode.DOWN, KeyCode.F4); ①
assertThat(commander.getMode()).isEqualTo(Mode.EDIT);

type("Roadmap: ");
press(KeyCode.END);
type("!");
assertThat(commander.isModified()).isTrue();
screenshot("editor");

press(KeyCode.F2); ②
assertThat(commander.getStatus()).isEqualTo("Saved plan.md");
assertThat(Files.readString(work.resolve("plan.md"))).isEqualTo("Roadmap: # Plan!\r\n\r\n1.
Draw\r\n2. Run\r\n3. Bind\r\n"); ③

press(KeyCode.ESCAPE);
assertThat(commander.getMode()).isEqualTo(Mode.BROWSE);
```

① Edit `plan.md`.

② Save.

③ The file had Windows line endings, and it still has them.

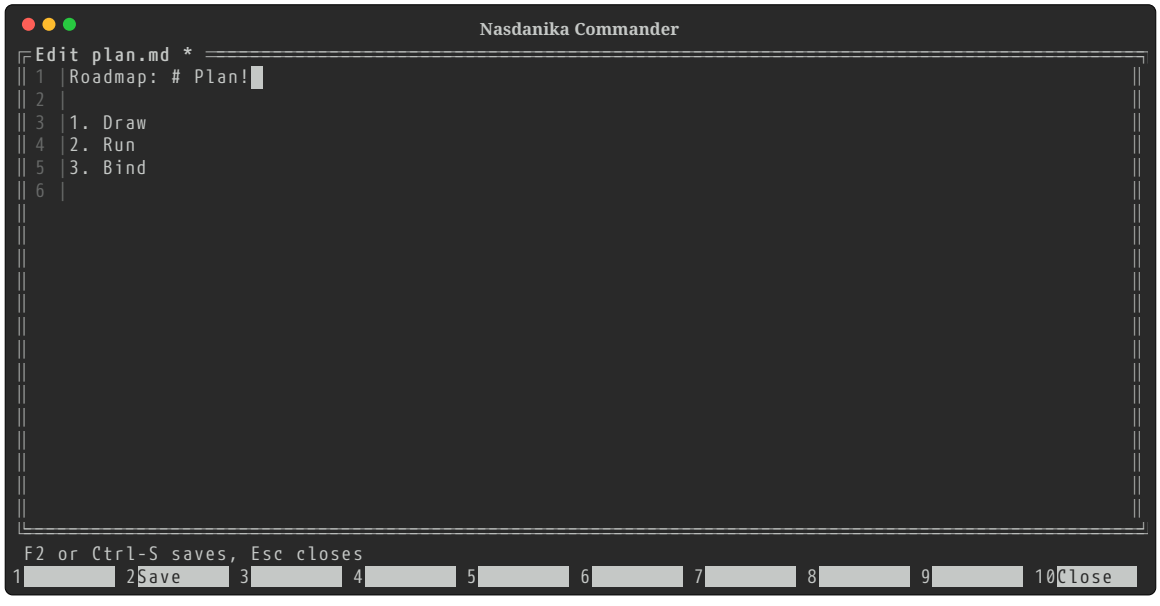


Figure 4. The editor, with a change not yet saved

The editor is deliberately small, and three rules keep it safe.

```
private void openText(Entry entry, boolean edit) throws IOException {
    if (entry == null || entry.kind() == Entry.Kind.DIRECTORY) {
        status = "Select a file first";
        return;
    }
    String name = Volumes.name(entry.path());
    if (entry.size() > MAX_TEXT) {
        status = name + " is too large to open here";
        return;
    }
    String decoded;
    try {
        decoded = decode(Volumes.read(active.getVolume(), entry.path())); ①
    } catch (CharacterCodingException e) {
        status = name + " is not UTF-8 text"; ②
        return;
    }
    lineSeparator = decoded.contains("\r\n") ? "\r\n" : "\n"; ③
    boolean writable = active.isWritable();
    startText(active, entry.path(), decoded.replace("\r\n", "\n"), edit && writable);
    if (edit && !writable) {
        status = "This panel is read-only: viewing";
    }
}
```

① The whole file is read through the volume.

② A file that is not UTF-8 text is not opened: an editor that guesses an encoding damages files.

③ The line endings are remembered and put back on save.

```
private void save() throws IOException {
    if (!newFile && !isModified()) {
        status = "No change, nothing written"; ①
        return;
    }
    GenericVolume.Writable<Content> writable = Volumes.writable(textPanel.getVolume
()).orElseThrow(() -> new IOException("The volume is read-only"));
    String current = text.text();
    writable.write(textPath, Content.of(current.replace("\n", lineSeparator).getBytes
(StandardCharsets.UTF_8))); ②
    original = current;
    newFile = false;
    status = "Saved " + Volumes.name(textPath);
    refresh();
    textPanel.select(Volumes.name(textPath));
}
```

① Saving without a change writes nothing, so a file that was only looked at keeps its bytes

and its date.

② The volume writes. Over a directory it writes to a temporary file and moves it over the target, so a failure leaves the file as it was.

A file is created with `Shift` + `F4`, and it exists only once it is saved:

```
commander.handle(KeyEvent.ofKey(KeyCode.F4, KeyModifiers.SHIFT)); ①
type("ideas.txt");
press(KeyCode.ENTER);
assertThat(commander.getMode()).isEqualTo(Mode.EDIT);
assertThat(work.resolve("ideas.txt")).doesNotExist(); ②

type("screens drawn in Draw.io");
press(KeyCode.F2);

assertThat(Files.readString(work.resolve("ideas.txt"))).isEqualTo("screens drawn in Draw.io");
```

① Shift and F4, as a terminal reports it.

② Named, and open in the editor, but not on disk yet.

8.5. Copy, move and delete

```
press(KeyCode.DOWN, KeyCode.F5); ①
assertThat(screen(100, 24)).contains("Copy plan.md to, in archive:").contains("plan.md");
screenshot("copy");

press(KeyCode.ENTER); ②

assertThat(commander.getStatus()).isEqualTo("Copied 1 file to plan.md");
assertThat(archive.resolve("plan.md")).hasSameBinaryContentAs(work.resolve("plan.md"));
assertThat(work.resolve("plan.md")).exists();
```

① Down to `plan.md`, and copy. A dialog asks where to, with the other panel's directory filled in.

② Enter accepts it.

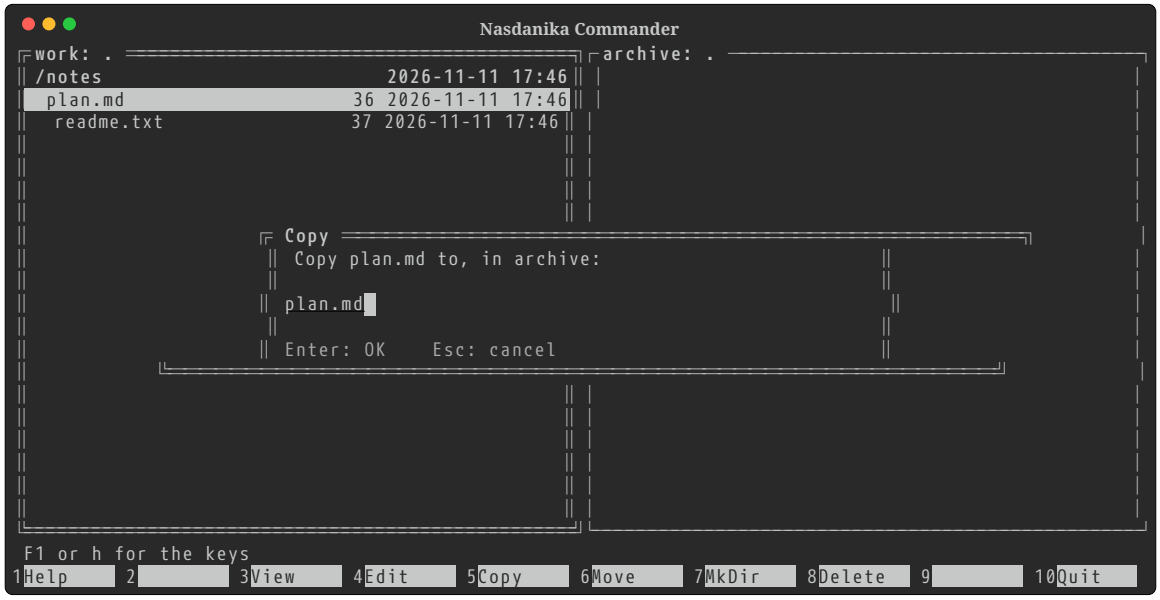


Figure 5. Copying to the other panel

```
press(KeyCode.DOWN, KeyCode.DOWN, KeyCode.F6); ①
for (int i = 0; i < "readme.txt".length(); ++i) {
    press(KeyCode.BACKSPACE);
}
type("about.txt"); ②
press(KeyCode.ENTER);

assertThat(commander.getStatus()).isEqualTo("Moved readme.txt to about.txt");
assertThat(work.resolve("readme.txt")).doesNotExist(); ③
assertThat(Files.readString(archive.resolve("about.txt"))).isEqualTo("A directory to try the
commander on.\n");
```

① Down to `readme.txt`, and move.

② The name in the dialog is replaced, so the move is also a rename.

③ Gone from the source.

```
press(KeyCode.F8); ①
assertThat(screen(100, 24)).contains("Delete the directory notes and everything in it?");

press(KeyCode.ESCAPE); ②
assertThat(work.resolve("notes")).exists();

press(KeyCode.F8, KeyCode.ENTER);
assertThat(work.resolve("notes")).doesNotExist(); ③
assertThat(commander.getStatus()).isEqualTo("Deleted notes, 2 entries");
```

① Delete, on a directory.

② Escape answers no.

③ Enter answers yes, and the directory goes with what was in it.

A target that exists is asked about before it is replaced, and an edited file is asked about before it is closed.

The file work is written against the volume SPI and nothing else:

```
public static int copy(Volume source, String path, Volume target, String targetPath) throws
IOException {
    GenericVolume.Writable<Content> writable = writable(target).orElseThrow(() -> new
IOException("The target volume is read-only"));
    Entry entry = source.stat(path).orElseThrow(() -> new IOException("Not found: " + path));
    if (entry.kind() != Entry.Kind.DIRECTORY) {
        writable.write(targetPath, source.content(path)); ①
        return 1;
    }
    if (source == target && (targetPath.equals(path) || targetPath.startsWith(path + "/"))) {
        throw new IOException("Cannot copy a directory into itself");
    }
    if (target.stat(targetPath).isEmpty()) {
        writable.createDirectory(targetPath);
    }
    int copied = 0;
    for (Entry child: source.list(path)) { ②
        copied += copy(source, child.path(), target, child(targetPath, name(child.path())));
    }
    return copied;
}
```

① Content goes from one volume to the other. Neither needs to be a directory on disk.

② A directory is copied with what is in it.

```
public static void move(Volume source, String path, Volume target, String targetPath) throws
IOException {
    if (source == target) {
        try {
            writable(source).orElseThrow(() -> new IOException("The volume is read-only")).move
(path, targetPath);
            return;
        } catch (IOException e) {
            if (source.stat(path).isEmpty() || source.stat(targetPath).isPresent()) {
                throw e; ①
            }
        }
    }
```

```
    copy(source, path, target, targetPath); ②
    delete(source, path);
}
```

① A move that failed for a reason of its own, such as a target that exists, is reported.

② Between volumes, and where a volume cannot move, a move is a copy and a delete.

8.6. A panel is a volume

To show something in a panel, make it a volume. A read-only volume needs four methods. This one, from the tests, shows the modules of the running JVM, one text file each:

```
class ModulesVolume implements Volume {

    private final Map<String, Module> files = new TreeMap<>();

    ModulesVolume() {
        for (Module module: ModuleLayer.boot().modules()) {
            files.put(module.getName() + ".txt", module);
        }
    }

    private static byte[] describe(Module module) {
        StringBuilder text = new StringBuilder("module ").append(module.getName()).append('\n');
        module.getDescriptor().requires().stream().sorted().forEach(requires -> text.append("
requires ").append(requires.name()).append('\n'));
        return text.toString().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public Optional<Entry> stat(String path) {
        if (".".equals(path)) {
            return Optional.of(new Entry(".", Entry.Kind.DIRECTORY, -1, null, null));
        }
        Module module = files.get(path);
        return module == null ? Optional.empty() : Optional.of(new Entry(path, Entry.Kind.FILE,
describe(module).length, null, Instant.EPOCH));
    }

    @Override
    public List<Entry> list(String directory) {
        List<Entry> entries = new ArrayList<>();
        for (String file: files.keySet()) {
            entries.add(stat(file).orElseThrow());
        }
        return entries;
    }

    @Override
```

```
    public Content content(String path) {
        return Content.of(() -> {
            Module module = files.get(path);
            if (module == null) {
                throw new NoSuchFileException(path);
            }
            return new ByteArrayInputStream(describe(module));
        });
    }

    @Override
    public boolean caseSensitive() {
        return true;
    }

    @Override
    public <T> Optional<T> capability(Class<T> type) {
        return Optional.empty(); ①
    }

}
```

① No capabilities, so it is read-only, and the commander finds that out by asking.

```
Volume modules = new ModulesVolume(); ①
commander = new Commander(new Panel("jvm", modules, "."), new Panel("archive", Volume.of(
archive), "."));
commander.getLeft().select("java.base.txt");

press(KeyCode.F4); ②
assertThat(commander.getMode()).isEqualTo(Mode.VIEW);
assertThat(commander.getStatus()).isEqualTo("This panel is read-only: viewing");
assertThat(screen(100, 24)).contains("module java.base");
press(KeyCode.ESCAPE);

press(KeyCode.F8);
assertThat(commander.getStatus()).isEqualTo("This panel is read-only");

press(KeyCode.F5, KeyCode.ENTER); ③
assertThat(Files.readString(archive.resolve("java.base.txt"))).startsWith("module java.base");
screenshot("jvm");
```

① The left panel is the JVM.

② Editing a read-only volume becomes viewing, and says so.

③ Copying **from** it works: a module's description is now a file in a directory.

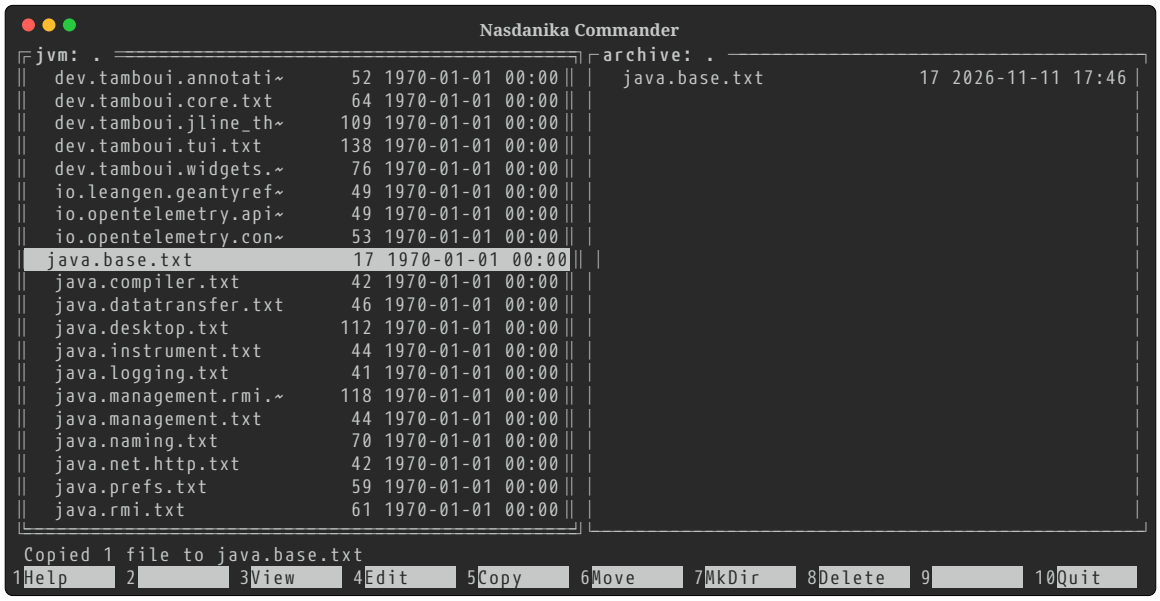


Figure 6. The modules of a JVM on the left, a directory on the right

Several volumes become one by mounting them under names, which is also how the drives of a Windows machine appear in one panel:

```
Map<String, Volume> mounts = new LinkedHashMap<>();
mounts.put("jvm", new ModulesVolume());
mounts.put("work", Volume.of(work));
Volume mounted = new MountVolume(mounts); ①

commander = new Commander(new Panel("mounted", mounted, "."), new Panel("archive", Volume.of
(archive), "."));
assertThat(commander.getLeft().getEntries()).extracting(entry -> entry.path()).containsExactly
("jvm", "work");

press(KeyCode.DOWN, KeyCode.ENTER, KeyCode.DOWN, KeyCode.DOWN, KeyCode.F4); ②
assertThat(commander.getMode()).isEqualTo(Mode.EDIT);
type("> ");
press(KeyCode.F2, KeyCode.ESCAPE);

assertThat(new String(Files.readAllBytes(work.resolve("plan.md")), StandardCharsets.UTF_8
)).startsWith("> # Plan");
```

① A volume of two mounts, one read-only and one writable.

② Into `work`, down to `plan.md`, and edit. Writing goes to the mount the path names.

8.7. Through a real terminal

The tests above call the application directly. One test runs it in its loop, on a JLine terminal that exists only in memory, and sends the bytes a terminal sends for the keys:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
terminal = new LineDisciplineTerminal("commander", "xterm-256color", out, StandardCharsets.
UTF_8); ①
terminal.setSize(new Size(100, 24));

Volume volume = Volume.of(directory);
Commander commander = new Commander(new Panel("left", volume, "."), new Panel("right", volume,
"."));
TuiConfig config = TuiConfig.builder().backend(new JLineBackend(terminal)).shutdownHook(false
).build();

AtomicReference<Throwable> failure = new AtomicReference<>();
Thread loop = Thread.ofPlatform().daemon().name("commander").start(() -> {
    try {
        CommanderMain.run(commander, config); ②
    } catch (Throwable e) {
        failure.set(e);
    }
});

send(SHIFT_F4); ③
await("the dialog that asks for a name", commander::isDialogOpen);
send("hello.txt" + ENTER);
await("the editor", () -> commander.getMode() == Mode.EDIT);
send("typed in a terminal");
send(F2);
await("the file", () -> Files.exists(directory.resolve("hello.txt")));
send(F10); // Closes the editor
await("the panels", () -> commander.getMode() == Mode.BROWSE);
send(F10); // Quits
loop.join(TimeUnit.SECONDS.toMillis(15));
```

① A terminal of 100 by 24 whose screen is a byte buffer and whose keyboard is the test.

② The same loop that `main` runs.

③ `ESC [ 1 ; 2 S`: F4 with the shift modifier, as xterm sends it.

So the classic keys are known to arrive as the keys they are, at least in the encoding of xterm.

8.8. What is not there yet

A list for a tool that is meant to be used.

- PLANNED **Marking several entries,** with `Insert`, and acting on all of them.
- PLANNED **Progress and cancel** for a long copy, and keeping the time of last change on a copy.
- PLANNED **Search** in a panel and in the viewer, and a viewer for files that are not text or are too large to hold.
- PLANNED **A navigator with details:** one panel and, beside it, what the selected entry is, instead of a second panel. For a model that is the more useful form.
- PLANNED **Any tree, with commands by type.** A panel over a generic volume of model elements or of the pages and shapes of a Draw.io file, where the commands offered depend on what is selected. Navigating a structure and acting on its elements is the polymorphic user interface in another form.
- PLANNED **Extensions through the capability framework:** syntax highlighting, commands, and an assistant that can be asked about what is selected.
- PLANNED **A command for scripts and agents** that runs an action on a path, so that what a person can do with a key, an agent can do with a line.
- PLANNED **As a command of `nsdk`,** with the volumes that the commands before it on the line produce, and in an organization's own command line with its own volumes.

Chapter 9. Screens Drawn in Draw.io

Capability	A user interface drawn before it is programmed: screens and the flow between them as pages of a Draw.io file, which runs with nothing connected, and to which handlers are then bound one at a time. Drawn once, shown in a terminal, on a page and in a document.
Demonstration	None yet. The first drawing is to be the commander of Chapter 8.
Status	PLANNED. Nothing in this chapter has been built or run.
Depends on	Chapter 6 and Chapter 7 for what a drawn element becomes, Chapter 4 for what it shows, and the runtime's Draw.io model for reading the file.
Value and effort	High for those who think in pictures, and unknown in size: the interpreter may be larger than it looks.
How to help	Draw a screen flow of your own with plain Draw.io shapes and say what you needed that stacks and containers did not give you. That is the first unknown.

9.1. Why

Some people see a system before they can describe it. I am one of them: schematics, CAD, integration flows drawn on a canvas, user interfaces assembled in a visual builder. In all of those the drawing was the program and not a picture of it.

For a user interface that means seeing the screens and the way between them first, and connecting behavior second. It also means that the people a screen is for can see it, and say what is wrong with it, long before it runs, and some of them never will see it run.

Draw.io is the editor because people already have it, and because a `.drawio` file is a model the runtime already reads. There is no editor to build.

This continues earlier work: wireframes of web pages drawn in Draw.io with a library of Bootstrap shapes, from which a site was generated and which were also used when the application ran. The terminal is a better place to start again. Its vocabulary is small, a cell grid forgives a rough drawing, and there are few tools of this kind for it.

9.2. Draw, run, then bind

The order of the work follows the way of working.

Step	What a person does	What it needs
Draw	The flow and the screens, as pages of a file, with links between pages	A library of shapes, and no Java
Run	Opens the file in a terminal and walks through it with nothing connected	An interpreter
Bind	Connects one handler at a time, against a list of what is not connected yet	Selection annotations and a validator
Change	Edits the drawing and sees the screen follow	Reloading on save

Three things follow.

Unbound is a state and not a failure. A screen with no handlers still runs. Elements show their labels or sample text, and an action that is not connected says so on the status line. That is the experience of a visual builder: a user interface exists before any logic does.

The validator has two modes. A handler for an element that is no longer in the drawing, two handlers for one element, and types that cannot fit are always errors. Elements with no handler are the to-do list while working, and errors only in a strict mode, for a build or a release.

The library comes first. It can be drawn with before a line of the interpreter exists, and what people draw with it says what the interpreter has to do.

9.3. What is drawn

Screens. A page is a screen. Draw.io's horizontal and vertical stacks and its containers give the structure: what is inside what, and in what order. Whether they carry enough structure for real screens is the first thing a trial has to answer. How a list, or anything else whose content is not known when it is drawn, is expressed is the second.

Shapes are kinds. A shape in the library stands for a kind of Chapter 6: text, a link, a section, a list, a table, and later an action and an input. Two libraries are planned. A generic one has the kinds every surface has, with the same names as in the widget module. A terminal one adds what only a terminal context has, marked as such, so that a drawing made for a terminal says so.

Flows. A link from an element to a page is navigation: the button opens that screen, in the whole terminal, in a dialog or in a panel. A screen may also define actions, such as "Next", whose meaning is given by a flow diagram that connects screens. Behavior that is not user interface is drawn there.

Reusable screens. A page can be used inside another, as a panel or a dialog. And a drawing can belong to a class of a model: the module that provides a credit card account provides the drawing of its screen, and a screen that shows an account finds it, as views are found by class everywhere else in the runtime.

Styles. A drawing's colors map to logical colors, and those to whatever the surface has.

9.4. How a drawing meets Java

Two things are connected to an element of a drawing, and the roadmap keeps two words for them.

An element is **resolved** to what implements it: a method that, given the element, returns the widget or the action for it. That goes through the capability framework, with a selection annotation that says which document, page and element a method is for: by id, by label, or by the value of a property. The annotation is what lets a validator find a method whose element is gone.

An element is **bound** to a value: a binding, which it shows and follows, or which it writes when the user types.

A third word, **wire**, is for a connection drawn between two elements, where one follows the other.

The mapping is one way at first. Java says which elements it implements, and a validator and the interpreter read the drawing and connect it. Handlers written inside the drawing, in a script, wait until they are asked for, and would be off unless a drawing comes from a trusted place.

```
@DiagramResource("card-account.drawio")                          ①
public class CardAccountScreens extends DiagramCapabilityFactory {

    @Capability
    @Element(id = "payment-due")                                  ②
    public Function<CreditCardAccount, Widget<?, ?>> paymentDue(Node element) {
        return account -> ...;
    }

}
```

① The drawing these methods are for.

② The element this method implements.

The listing is a design and was never compiled. The names of the annotations are open: `@Document`, `@Page` and `@Element` read best, and the first two are also the names of classes in the Draw.io model.

9.5. One drawing, several surfaces

A drawing made of generic shapes names kinds and not a format. So the screen that runs in a terminal can also be a static page with Bootstrap, an application in React, and a figure with a description in a document.

The last one closes a loop with Chapter 6. Draw.io is itself a surface: a widget context that writes shapes onto a page would draw a screen from a view, and the result could be opened, corrected and used as the drawing from then on.

Documentation comes from the same file: the screens as figures, the flow, who owns each screen and what state it is in.

9.6. Usage against design

The drawn flow is the set of paths that were designed. With a span per screen and per action, as Chapter 4 describes, telemetry has the paths that people took. Laying the counts of the second over the first shows the paths nobody uses and the detours everybody takes.

9.7. Prior art

Visual builders for desktop user interfaces, where a form is drawn and handlers are attached. Layout files of mobile platforms, where a screen is a resource and code finds its parts by id. Builders for terminal interfaces exist as editors of their own that export code for one framework or another; a survey in November 2026 found none that reads a Draw.io file or runs the file itself, which is weak evidence that the combination is new and no evidence that it is wanted. The honest position is that this is built for its author first.

Chapter 10. Designed and Waiting

The items in this chapter are designed, some of them in detail, and none has code in this repository yet. Each will get a chapter when it gets a module. They are grouped by area, and within a group roughly by value for effort as it looked in November 2026.

10.1. Toward the North Star

These are the increments that lead to agentic work on typed models, and they will be shown in the Nasdanika International Banking System.

Item	What it is
PLANNED Inference	A model of inference, with providers called over the JDK's HTTP client: a conversation, its turns and its tool calls are a model that can be saved, loaded and continued with another provider. Chapter 12 has the sketch.
PLANNED Tools over volumes	File tools for an agent, working on a volume, and an overlay in memory as a sandbox: the agent writes, and what it wrote is reviewed before it reaches the volume below.
PLANNED MCP	A server and a client on one JSON-RPC core, which a language server can share. Tools come from the tool model.
PLANNED The change model as the command	Agents read with queries and write with changes. A change is data, so it can be validated, shown and approved before it is applied. Chapter 11 has the sketch.
PLANNED State in Git	A commit as a message between the steps of an executable graph, and Git as its durable log. Chapter 5 has the design of the carrier.
PLANNED AI as an operator	A model provider as a step of rendering or transformation: a binding that runs once its input has been still for a while in a live screen, and once, cached by a hash of its input and labeled as generated, in a document.

10.2. Seeing and changing a running system

Item	What it is
PLANNED Live telemetry	The spans of a unit of work as they happen, over server-sent events in a browser and as a tree in a terminal.
PLANNED System volumes	A system presented as a volume of models, so that it can be browsed, copied as a snapshot and compared with the next one. A running JVM first, then the resources of a cloud account.
PLANNED A sanitizer for telemetry	Values mapped before they reach logs and telemetry: as they are for a test, masked for production, encrypted or replaced by tokens. The mapping is chosen by what a value is, from classifications in the model, and its strength by the environment.
PLANNED Model auto-save	A command that changes a model in place saves what it changed when it succeeds and validates. It is built on change recording: only net changes are saved, everything is serialized before anything is written, and a failed write rolls the model back.
PLANNED Reactive compute graphs	Processors connected by Chapter 4, settling by the lineage of Chapter 5. The first practical graph has arithmetic in its nodes: cost rolled up and risk propagated along the dependencies of an enterprise model. Nodes that call a model provider come after, paced by their inputs and cached by them.

10.3. The command line

Item	What it is
PLANNED Commands over what is built	`drawio render` and `drawio video`, SVG and diagram generation, and `http`, `https` and `webdav` to serve a volume. The handlers and the APIs exist, and the commands are small.
PLANNED `nsdk install` and profiles	An installation extended from Maven coordinates, through the mirrors an organization has configured and nothing else, with profiles that compose.
PLANNED Self-documenting command lines	The command tree that a set of modules assembles, saved as a model and rendered by Chapter 6 into a site, a book or a terminal browser, and compared between releases. Chapter 12 has the model.

Item	What it is
PLANNED An edit command	The editor of Chapter 8 as a command of its own, for a file, an attribute of a model element, or a file in a repository on a server.

10.4. Documents and diagrams

Item	What it is
PLANNED Generated metamodel references	The reference of a model generated into its book from its package, with the hand-written text in files that are created once and never overwritten, and a check in the build that the reference is current.
PLANNED The document model and a Markdown reader	Structure in the model and text in a named format. Markdown is read into it, and AsciiDoc, Markdown and HTML are written from it. Chapter 11 has the sketch.
PLANNED Measured text for generated diagrams	A table of character widths for one font, so that a diagram laid out on one machine looks the same on every other.
PLANNED Draw.io actions and animation	Custom actions of a Draw.io diagram as objects, and from them animations recorded as video: a trace shown as a diagram that moves.
PLANNED Video as a build artifact	A model as the script of a video, narration anchored to its words, recorded with the browser automation the runtime has, and assembled by the build. No timeline editor.
PLANNED Office formats from their schemas	The package format and the word processing format first, read into models generated from the published schemas, without a third-party library in between.
PLANNED A merger	Merging generated files with files that people changed: create once, keep a sidecar, or merge three ways. Draw.io diagrams are merged by element id.
PLANNED Content digests	A digest of a model element computed from its content, as an identifier that does not depend on where the element is, and later signatures over digests.
PLANNED The polymorphic user interface in a browser	React with Bootstrap, after the terminal. A view covers the features it shows and hands the rest to a less specific view, so that the reflective view at the end of the chain shows only what nobody claimed.

10.5. Books and the body of knowledge

Item	What it is
PLANNED The runtime’s site	A home page generated from the build’s own description of its modules, with a link to each module’s book.
PLANNED The books in the IDE	An extension that brings the books of the installed version into the editor, where an assistant can read them too.
PLANNED The International Banking System book	The practice of architecture as code, taught on one example that grows with the platform.
PLANNED Cascading roadmaps and architectures	A roadmap that depends on another team’s released roadmap, as a build depends on a library: a slip shows up as a difference, and conflicting dates as warnings. The same for architecture repositories. This roadmap, with ids and dates as data, would be the first to be depended on.

Part III: Models

The models on the roadmap: the nano-models that are to join the core model, the models of the runtime itself, the model tower, the executable graph and mapping models to revisit, and the technology models to migrate.

Chapter 11. Core Nano-models

Capability	Small aspects that many models share, defined once in the core model: relationships, changes, occupancy in time and other dimensions, kinds, stakeholders and what they are shown, documents, and examples.
Demonstration	`NanoModelTests` in `models/core`, which instantiates each one.
Status	`SKETCH` in `models/core`: seven Xcore files that generate and compile against the released core model. Role, IAM, Seal and Lifecycle are in `models/core/sketches` as they are in the tower today.
Depends on	The runtime's core model.
Value and effort	High, and each is small. Every model above the core gets them for nothing, and several models in the tower shrink when they arrive.
How to help	Read the one closest to your work and say which word is wrong. Naming is the hard part of a nano-model and the part that cannot be changed after release.

11.1. What a nano-model is

A Maven module can hold many Xcore models. A nano-model is one file: an aspect small enough to read at once, in a package of the core module, with a namespace of its own. It is in core because nearly every model needs it and because it needs nothing that core does not have. A model with other dependencies, other users or another pace of change is a module of its own, which the SDK calls a micro-model.

The nano-models depend on each other in a line, so there is an order in which to read them and no cycle to untangle.

Nano-model	Builds on	In one line
related	core	Relationships as model elements

Nano-model	Builds on	In one line
change	core	Changes as data, with selectors and computed values
occupancy	related, change	A subject occupies a box in dimensions, of which time is one
kind	change, occupancy	Classification as data, which can configure
enterprise	core	Stakeholders, concerns, viewpoints, formats and channels
document	enterprise	Structure in the model, text in a named format
example	core	An input, the output expected for it, and why

Where a later nano-model needs a class of an earlier one to say more, it specializes that class, and only that class, and only when something needs it. A role model that needs documents with authors specializes `Section`, and leaves the rest of the document model alone.

The pattern

Most nano-models have the same four parts, shown here by the kind model:

```
/**
 * An element classified with kinds, in an order that matters when kinds configure.
 */
interface Kinded {

    contains AbstractKind[] kinds

}

/**
 * What a list of kinds holds: a kind defined in place, or a reference to one in a catalog.
 */
abstract class AbstractKind extends ModelElement {

}

class Kind extends AbstractKind, Changing {

    String name

    /** The more general kinds. Several are allowed. */
```

```
    refers Kind[] specializes

}

class KindReference extends AbstractKind {

    refers Kind target

}

class KindCatalog extends NamedElement {

    contains Kind[] kinds

    contains KindCatalog[] catalogs

}
```

`AbstractKind`

What a list holds: either of the two below.

`Kind`

The definition.

`KindCatalog`

Definitions published together, as a resource that others refer to.

`KindReference`

A use of a definition from a catalog, in a place where a definition could also be written.

The reference class was designed before one fact about EMF was put to use here: containment can cross resources. A contained object may live in another file, so splitting a model over files does not need a reference class. A reference class stays where the user of a definition has something of its own to say about the use, and it may go from the nano-models where it has not.

11.2. Related

A reference says that two elements are related. A relationship says so and can be asked about, because it is a model element: it has an identity, documentation and provenance, and other nano-models give it more. `Related` is the interface of an element that has such

relationships.

The architecture model of the tower is mostly this: elements, containment and relationships that are objects. With `Related` in core, little of it is left to define.

11.3. Change

EMF Change describes a change by pointing at the objects it changed and holding the values it set. The change nano-model keeps the idea and changes two things. A change finds its objects with a **selector**, an evaluator run in the context of a host. And a new value can be **computed** from that context.

So a change can be written before the objects it applies to exist, kept in a catalog, and applied to many hosts. Two nano-models need exactly that: a kind applies its changes to whatever it classifies, and a transition applies its changes to what moves.

A change is a plan. It can be validated, shown to a person and approved before anything is applied, which is why it is also the command an agent should write with: read with a query, write with a change. Applying a change is recorded by EMF's change recorder, whose description is the reverse of what was done.

Three rules come with it. Changes apply in order, and each sees the result of those before it. A change sets values in the structure its host has and never adds to that structure. And selectors and values have no side effects, with XPath as the language for catalogs that are not trusted to run scripts.

11.4. Occupancy

Most models treat time as two date columns. Occupancy treats it as one dimension among others: a jurisdiction, a version line, a level of sensitivity, a floor of a building.

A **region** is a box, with at most one extent per dimension. An **occupancy** is the recorded fact that a subject is in a region. Plans, the present and history are then one model read at different coordinates: a person to be hired next year, a control that applies only in one jurisdiction, a price valid for a quarter.

```
Dimension time = OccupancyFactory.eINSTANCE.createDimension();
time.setName("Calendar time");
time.setComparator("iso8601");

Occupancy draft = occupancy(time, "2026-11-01", "2026-11-11"); ①
Occupancy published = occupancy(time, "2026-11-11", null); ②
published.getPredecessors().add(draft);

Transition publish = OccupancyFactory.eINSTANCE.createTransition(); ③
publish.setName("publish");
publish.setTarget(published);
SetChange status = ChangeFactory.eINSTANCE.createSetChange();
status.setFeature("status");
status.setLiteral("published");
publish.getChanges().add(status);
draft.getRelationships().add(publish);

assertThat(publish.getSource()).isSameAs(draft);
assertThat(published.getExtents().get(0).getEnd()).isNull();
assertThat(draft.getExtents().get(0).isEndExclusive()).isTrue();
```

① A draft, from the first of November to the eleventh. The end is exclusive, and the model says so.

② Published from the eleventh, with no end: unbounded, not unknown.

③ A transition from one occupancy to the other, with the change it makes to what moves.

The rules are in the header of `occupancy.xcore`, and four of them prevent most mistakes. A dimension that a region does not mention is unconstrained. Whether a bound is inclusive is stored and never assumed. A shape is a union of boxes, two occupancies and not a polygon. And an occupancy is a record: a correction is a new occupancy that names the one it replaces.

`Temporal` and `Period`, which the older core model had as special cases, become a coordinate and an extent on the calendar dimension. Stages of a lifecycle, presence in a place and participation in an engagement are occupancies with one more reference each.

11.5. Kind

A class says what an element is, for good. A kind says what it is classified as, and that can be one of several, can come from a catalog that someone else publishes, and can stop applying.

A kind can also configure. It carries changes, and they apply to what it classifies: the size of a machine, from a vendor's catalog, sets the memory of every machine classified with it.

```
SetChange memory = ChangeFactory.eINSTANCE.createSetChange(); ①
memory.setFeature("memory");
memory.setLiteral("16");
memory.setMode(ChangeMode.CONSTRAINT);

Kind large = KindFactory.eINSTANCE.createKind();
large.setId("large");
large.setName("Large");
large.getChanges().add(memory);

KindCatalog catalog = KindFactory.eINSTANCE.createKindCatalog();
catalog.setName("Machine sizes");
catalog.getKinds().add(large);

KindReference reference = KindFactory.eINSTANCE.createKindReference(); ②
reference.setTarget(large);
Classification classification = KindFactory.eINSTANCE.createClassification();
classification.setKind(reference);

ResourceSet resourceSet = resourceSet();
Resource catalogResource = resourceSet.createResource(URI.createFileURI(directory.resolve
("sizes.xmi").toString()));
catalogResource.getContents().add(catalog);
Resource machineResource = resourceSet.createResource(URI.createFileURI(directory.resolve
("machine.xmi").toString()));
machineResource.getContents().add(classification);
catalogResource.save(null);
machineResource.save(null); ③

Resource loaded = resourceSet().getResource(machineResource.getURI(), true); ④
Classification loadedClassification = (Classification) loaded.getContents().get(0);
Kind loadedKind = ((KindReference) loadedClassification.getKind()).getTarget();

assertThat(loadedKind.getName()).isEqualTo("Large");
assertThat(loadedKind.eResource().getURI().lastSegment()).isEqualTo("sizes.xmi");
assertThat(((SetChange) loadedKind.getChanges().get(0)).getLiteral()).isEqualTo("16");
```

① A change that sets a feature, as a constraint: a host that says otherwise is in error.

② A use of the kind, by reference, in a classification.

③ The catalog and its user are saved as two files.

④ Loading the user finds the kind in the catalog's file.

The boundary is one sentence. A kind sets values in the structure its host has and never adds to it; a kind that wants a feature its host's class does not have wants to be a class.

Where kinds specialize several others, the order of their changes is the C3 linearization, the one Python uses for classes. And changes are layers over the host and not edits of it, so that ending a classification removes a layer and nothing has to be undone. Neither is built yet.

11.6. Enterprise

Every effort has stakeholders with concerns, and few write them down. The enterprise nano-model is the first model an effort should make: who the stakeholders are, what concerns each, what they are shown to address it, in what format, and over what channel.

The vocabulary is that of ISO/IEC/IEEE 42010, with formats and channels added.

```
Format markdown = format("markdown");
Format html = format("html");
Format pdf = format("pdf");
Channel email = channel("E-mail", html, pdf); ①
Channel mcp = channel("MCP", markdown);

Stakeholder customer = EnterpriseFactory.eINSTANCE.createStakeholder();
customer.setName("Customer");
customer.getFormats().addAll(List.of(pdf, html)); ②
customer.getChannels().add(email);
Concern finances = EnterpriseFactory.eINSTANCE.createConcern();
finances.setName("What is the state of my finances?");
customer.getConcerns().add(finances);

Stakeholder agent = EnterpriseFactory.eINSTANCE.createStakeholder();
agent.setName("Assistant");
agent.getFormats().add(markdown);
agent.getChannels().add(mcp);

Viewpoint statement = EnterpriseFactory.eINSTANCE.createViewpoint();
statement.setName("Monthly statement");
statement.setDocumentation("Opening balance, transactions by date, closing balance."); ③
statement.getFrames().add(finances);
statement.getElementKinds().add(DocumentPackage.Literals.TABLE); ④

assertThat(deliverable(customer, email)).containsExactly(pdf, html); ⑤
assertThat(deliverable(agent, mcp)).containsExactly(markdown);
assertThat(deliverable(agent, email)).isEmpty();
assertThat(finances.getStakeholder()).isSameAs(customer);
```

① A channel accepts formats.

② A stakeholder has formats in order of preference.

③ A viewpoint's documentation says how to read what is made from it.

④ A viewpoint names the kinds of things it is made of. A kind of representation element is the class of a representation model, here the table of the document nano-model.

⑤ What can be sent to whom is then computed: the formats that the stakeholder takes and the channel accepts.

The model is small because EMF already has most of the full enterprise model's kinds. A kind of semantic element is a class, a semantic element is any object, and a kind of representation element is a class of a representation model.

It has two uses. One is documentation: a "monthly statement" viewpoint that explains how to read a statement, a chart that explains itself, which matters most for something new. The other is a request. Asked for a viewpoint in a format over a channel, the polymorphic user interface builds the view, and the rendered view can open its viewpoint's documentation as help: a link on a page, `F1` in a terminal.

It also explains why the runtime has several formats for one picture. SVG is for people reading a page or a book, Draw.io for people who edit, and Markdown or the model itself for agents. Each is a row of a stakeholder, a channel and a format.

11.7. Document

The core model's `Documented` has sections and content. The document nano-model replaces them, and it starts from the enterprise nano-model: a document is a view, and a section is a representation.

```
Viewpoint statement = EnterpriseFactory.eINSTANCE.createViewpoint();
statement.setName("Monthly statement");

Document november = DocumentFactory.eINSTANCE.createDocument(); ①
november.setName("Statement for November 2026");
november.setViewpoint(statement);

Section transactions = DocumentFactory.eINSTANCE.createSection(); ②
transactions.setTitle("Transactions");
Text note = DocumentFactory.eINSTANCE.createText();
note.setText("No fees were charged in **November**.");
note.setFormat("markdown"); ③
transactions.getElements().add(note);
november.getRepresentations().add(transactions);
```

```
Message message = EnterpriseFactory.eINSTANCE.createMessage(); ④
message.setView(november);
message.setFormat(format("pdf"));

assertThat(november.getViewpoint().getName()).isEqualTo("Monthly statement");
assertThat(november.getRepresentations().get(0).getElements()).containsExactly(note);
assertThat(message.getView()).isSameAs(november);
```

① A document, made from a viewpoint, which a view always is.

② A section, with a title, in which sections and blocks nest.

③ Text stays in the format it was written in. The model owns structure, and the formats own text.

④ Delivering a document is a message: to whom, over what, in what format.

The model is a pivot. Markdown is read into it, and AsciiDoc, Markdown and HTML are written from it, so a new format costs one reader and one writer. A widget context of Chapter 6 can build it, and a document can be played back through the kinds into any other context.

11.8. Example

Examples are on tools, operations, mappings, prompts and viewpoints, so they are a nano-model and not a part of each.

```
Example example = ExampleFactory.eINSTANCE.createExample();
example.setName("A transfer between own accounts");
example.setInput(payload("{ \"from\": \"checking\", \"to\": \"savings\", \"amount\": 100 }"));
example.setOutput(payload("{ \"status\": \"completed\", \"fee\": 0 }"));
example.setRationale("Transfers between accounts of one customer have no fee.");

assertThat(Map.of(example.getInput().getFormat(), example.getOutput().getText()))
        .containsEntry("json", "{ \"status\": \"completed\", \"fee\": 0 }");
```

The same data has three uses. Examples teach, as a few-shot prompt or a section of documentation. An input with its expected output is a test case, so a published set of examples is a conformance suite for whatever they describe. And replaying the outputs runs a loop with no provider behind it.

11.9. Role, IAM, Seal and Lifecycle

Four models of the tower are to become nano-models: who is responsible, who may see and do what, what is signed and encrypted, and what stage something is in. They are the lowest floors of the tower, and everything above them inherits what they say.

Their Xcore is in `models/core/sketches`, as it is in the tower today, on the older core model. Moving them is a rewrite and not a rename:

- **Role.** An assignment of a role is an occupancy, and a role in a catalog is a kind. Sections with authors and editors specialize the section of the document nano-model.
- **IAM.** A grant valid in one region during one period is an occupancy. Levels of sensitivity are a dimension, so "cleared up to confidential" is an extent.
- **Seal.** Signatures are made over content digests, which are planned for core.
- **Lifecycle.** A stay in a stage is an occupancy, and a transition between stages is the transition of the occupancy nano-model, with its changes.

Each should be smaller afterwards than it is now, which is the test that the nano-models below it are right.

11.10. What is not there yet

- PLANNED **Applying changes:** the engine that evaluates selectors and values, applies changes in order, and reports a change that does not fit its host.
- PLANNED **Region arithmetic:** whether one box contains or meets another, and which occupancies are current at a coordinate.
- PLANNED **Catalogs as resources:** formats, channels and a few stakeholders that models can refer to, as the first content.
- PLANNED **Facets:** an element that has an aspect for a while, where today it would have to inherit it for good.
- PLANNED **Generated references** for these models, in this book, once the runtime generates them.

Chapter 12. Runtime Models

The runtime describes itself with models, as it describes everything else: a command line, a query, a tool an agent may call, a conversation with a model provider. The models in this chapter are PLANNED. Most have an Xcore file in `models/runtime`, to be read: it was written against the older core model, so it does not generate here, and each needs a pass against the runtime's core and the nano-models of Chapter 11 before it does.

Model	Pre-SDK Xcore	In one line
CLI	`cli.xcore`	A command line as data: commands as types, and the tree an installation resolves
Tool	`tool.xcore`	What can be called, apart from who calls it and how
Inference	`inference.xcore`	Providers and models as a catalog, and conversations as a record
Expression	`expression.xcore`	A small algebra of conditions that a data source can evaluate for itself
Cypher	`cypher.xcore`	The Cypher language as a model, for querying models as graphs
XPath	none yet	Paths through a model as selectors

12.1. CLI

Status PLANNED. The model is drafted in `models/runtime/cli.xcore`.

Depends on Meta, in the runtime, and Chapter 6 for rendering.

How to help Point it at a picocli application of your own once the loader exists. Until then, read the sketch against a command line you know.

The runtime's command line is not written in one place. Modules contribute commands, and the tree is assembled from whatever modules an installation has. So the command line is a property of an installation, and the only complete description of it is a snapshot of the resolved tree.

The CLI model is that snapshot. A command is a type, in terms of Meta: its options and

parameters are features with bounds, and an enumeration is an enumeration. A registration is one place in one tree where a command occurs, since one command can occur in many. A program is a dated snapshot with the modules it was resolved from.

Three uses follow.

- **Documentation.** The saved tree is rendered by Chapter 6 into whatever is needed: a site, a chapter, a page in a terminal. This is the successor of the HTML help that the older Nasdanika command line generated. The views for Meta render most of it, because commands are Meta classes.
- **Comparison.** Two snapshots are compared, as two models are, and the difference is what changed in a command line between two releases or two installations.
- **Tools.** A command described this way is a tool described in a schema, which an agent can be given.

The loader takes a root command and reads the tree through picocli's own API, so it works for any picocli application and not only for Nasdanika's. For command trees built on released picocli it waits for a picocli release that carries the description annotations the runtime contributed. The runtime's own picocli module has them already.

A Groovy DSL for pipelines of commands, and a form built from a command's options, are further out.

12.2. Tool

Status	PLANNED. Drafted in `models/runtime/tool.xcore`.
Depends on	Meta for schemas, and the example nano-model.

A tool definition says what can be called. Four models had begun to describe that separately: agents, MCP servers, inference requests, and the command line. The tool model is the one place.

It separates two things that are usually mixed. **Presentation** is how a language model learns about a tool and asks for it: native tool calling, or a prompt with examples whose result is validated against the same schema. **Invocation** is how the call happens: in

process, over a URI, through an MCP server. With the two apart, MCP is a transport and not a category, and a few-shot prompt is another calling convention for the same definition.

The model is useful with no language model in the picture. What a server exposes, who owns each tool, what it costs to call and whether it is approved is a catalog, and a catalog has readers of its own.

12.3. Inference

Status PLANNED. Drafted in `models/runtime/inference.xcore`.

Depends on The tool model, and the JDK's HTTP client for the providers.

The inference model has two halves.

The **catalog** is configuration: which provider, which model, which endpoint, what settings, what it costs and when it retires. Today that is in property files and constructor arguments, where it cannot be documented, compared, reviewed or owned.

The **session** is what happened: messages, tool calls, token counts and latency, as model objects. They are saved as JSON, opened in a viewer, kept for audit and redacted, on a laptop, with the data staying where it is.

Providers are modules over the JDK's HTTP client, with no vendor library: one for the Anthropic API and one for the OpenAI API, which also covers servers that speak it. What the tower called the AI runtime models is folded into this one.

Inference is the first increment on the path to the North Star of Chapter 1. A session saved with one provider and continued with another, chosen by a requirement rather than by name, is its first demonstration.

12.4. Expression

Status PLANNED. Drafted in `models/runtime/expression.xcore`.

A closed, small algebra of boolean conditions over values, made to be pushed down to a

data source. It is not an expression language. A construct belongs in it only if a data source could plausibly evaluate it for itself, and everything else stays with the caller, who applies it after the fetch. That fallback is always correct, so the model can stay small: expressiveness here buys speed and never correctness.

Cypher lowers into it, and so would a path language. It also separates syntax from semantics: how nulls compare and how mixed types are ordered is supplied by whoever uses it.

12.5. Cypher

Status PLANNED. The model is in `models/runtime/cypher.xcore`, the largest file there, as it is before the SDK.

Cypher as a model: a query is loaded into objects, which can be examined, rewritten and run.

The reason for Cypher is that it is known. People know it, and language models have seen a great deal of it, which cannot be said for a query language invented here. A model of typed objects with references is a graph, and a graph pattern is a natural way to ask it a question.

The plan has an order. A read-only subset comes first, published as a tool that is marked read-only from its first day. Writing goes through a separate tool that takes changes, of the change nano-model, and that accepts Cypher's updating clauses as one way to write them: they are translated into changes, shown as a plan, and applied after review. Agents read with Cypher and write with commands.

12.6. XPath

Status PLANNED. No file in this repository.

The change nano-model needs selectors, and catalogs of changes from untrusted sources need a selector language that cannot do harm. XPath is that language: it has no side effects, it cannot loop forever, and it is widely known. An XPath evaluator for model objects is the first step, beside the evaluators the core model has.

Chapter 13. The Model Tower

Capability	A set of small models of an organization and its solutions that build on each other, so that an element at the top is documented, owned, access controlled, staged, decided, governed, worked on and threat-modeled without any one model defining more than its own concern.
Demonstration	The Nasdanika International Banking System, as it grows. Today the tower is published as the pre-SDK models, each with its own site.
Status	PLANNED, floor by floor. The 22 models are in `models/tower` as Xcore files to read, as they are in the pre-SDK tower.
Depends on	Chapter 11, and Meta for the upper floors.
Value and effort	The largest item in this book, and the slowest. Its value is in composition, so it arrives a floor at a time.
How to help	Take the floor nearest to your work and model something real with it, on the pre-SDK version if need be. What a model cannot say is the most useful thing to learn before it is rebuilt.

13.1. What the tower is

The tower is not a product suite. It is a set of small models, each published on its own and each depending only on those below it. What makes it more than a pile is that the floors compose. A concern defined once near the bottom, such as identity, documentation, provenance or time, is inherited by everything above, and a model that takes its place in the tower stops defining what a lower floor already provides. The tower gets taller and the models get smaller.

Taken together they describe an organization from a button on a screen down to a field in a record of a mainframe, with ownership, access, stage, decisions, risks, controls, evidence, cost and work along the way. One graph then answers questions that today take four tools

and an export: which teams depend on this system of record, who approved this alternative and on what analysis, which controls cover this requirement, what breaks if this library has a vulnerability.

It is for anyone who has to understand something complicated quickly and pass it on accurately. A model is a deliverable that keeps working after the engagement that produced it: it can be handed over, extended, queried and regenerated. A reference architecture, a set of controls and a catalog of personas are models too, and new work starts from one of them and not from an empty page.

13.2. What changes with the SDK

The tower was built on the older Nasdanika, floor by floor, in Xcore. Three things change as it moves.

The lowest floors become the core. What the tower called NxCore is the runtime's core model. Time leaves it for the occupancy nano-model, where it is one dimension among others. Role, IAM, Seal and Lifecycle become nano-models. So the tower proper starts higher than it did, on a base that more models share.

Concepts move down, and models shrink. With relationships, kinds, occupancy and the enterprise vocabulary in core, several floors lose most of what they define. An architecture element with relationships is nearly `Related`. A persona with concerns is a stakeholder with concerns. A model is expected to be smaller after the move than before it, and where it is not, something below it is wrong.

The upper floors are defined in Meta. Up to lifecycle the models stay hand-written Xcore: a small, slow base that everything cites. From accounting upward a floor is defined in Meta, the runtime's language-neutral type system, and its Xcore, Ecore and TypeScript are generated. The floors above lifecycle are structure with almost no behavior, which is what Meta describes.

A product is built on a projection, not on the tower. Someone building a product for resumes needs a person, positions over time, organizations and skills, and not eight levels of supertypes. They select the elements they need, flatten the hierarchies they do not care about, and get a small model generated for them, in which every class and feature records

the tower element it came from. When the tower changes, the projection is generated again and compared, and an empty difference says the product is not affected. Large reference vocabularies have been cut down to subsets this way for twenty years. The tool that does it, an assembler, is PLANNED after the browser form of the polymorphic user interface.

13.3. The floors

The tables give each model with its file in `models/tower`, what it is, and what the move to the SDK does to it. Each row is a model to migrate, and none has a date.

The spine

Model	What it is	With the SDK
Accounting `accounting.xcore`	A light double-entry model: ledgers, commodities, accounts, entries, transactions and assertions about balances. Cost can be attributed to any element of the graph.	The first floor defined in Meta. Accounting periods are extents of the occupancy nano-model.
Decision analysis `decision-analysis.xcore`	Decisions with several criteria, several experts and several alternatives: judgments, direct weights, pairwise comparisons and results.	Deciders come from Role, confidentiality of judgments from IAM, and the workflow of a decision from Lifecycle.
Decision binding `decision-binding.xcore`	Which alternative was committed to which variation point, by whom, based on what analysis and what other decisions. Decision records that cannot drift from what they decide.	"Binding" here is a commitment, and in Chapter 4 it is a value. One of the two words has to give way.
Governance `governance.xcore`	Programs, frameworks, requirements, policies, controls, risks, evidence, assessments, findings and waivers, after the common shape of management systems.	Obligations get addressees from the enterprise nano-model.

Model	What it is	With the SDK
Work `work.xcore`	Work items attached to anything, with dependencies and constraints in time. A finding traces to the work that remedies it.	Time constraints are occupancies, including relative ones: "starts when the review clears".
Requirements `requirements.xcore`	Requirements, acceptance criteria, verification, and what satisfies what.	Satisfaction is a relationship, of `Related`.
Architecture `architecture.xcore`	Elements with containment, relationships that are objects, and kinds as data, so that C4, ArchiMate and TOGAF vocabularies load as catalogs.	Most of it is `Related` and the kind nano-model. What is left may be a catalog and not a model.
Threat `threat.xcore`	Assets, threats, weaknesses, trust boundaries and actors, with STRIDE, ATT&CK, ATLAS and the OWASP lists as catalogs. An asset is an architecture element seen for its security.	Being an asset becomes a facet held for a time, not a supertype. The Maven model then does not need to know that threats exist.
C4 `c4.xcore`	Person, software system, container, component and code, on architecture.	A catalog of kinds over `Related`, with the diagrams the runtime already draws.

Branches

Model	What it is	With the SDK
AI governance `ai-governance.xcore`	AI systems, model and dataset assets, risks, impact assessments, oversight controls, model cards, evaluations and incidents, grounded in the threat and governance floors.	Unchanged in intent. Not reviewed by a specialist in AI governance: it maps to public frameworks and does not claim to satisfy them.
Agent `agent.xcore`	Agentic systems, agents, tasks, tools and skills: for documenting such a system and for running it. An agent is an AI system, so it is governed by construction.	Runs on Chapter 5. Tools come from the tool model and inference from the inference model, so it should shrink. Authoring in Draw.io follows Chapter 9.

Model	What it is	With the SDK
MCP `mcp.xcore`	Servers, their versions, tools, resources and prompts, hosts and clients, and catalogs of them.	Tools move to the tool model, where MCP is one transport.
Capability `capability.xcore`	Capabilities, their providers, dependencies between them and evidence that a provider has what it claims.	Unrelated to the runtime's capability framework, which resolves requirements in code. The name needs care.
Product management `product-management.xcore`	Personas, their concerns, goals, needs and pain points, and the capabilities that concerns pull.	Personas and concerns move down to stakeholders and concerns of the enterprise nano-model.
Org design `org-design.xcore`	Organizations, units, roles and people, strategy and goals, processes, decision rights and coordination.	A person extends a stakeholder. Membership of a unit for two quarters is an occupancy.
Family `family.xcore`	Families, members, kinship, events and sources: genealogy, and a model anyone can check against what they know.	Kinship is a relationship with a kind, and a life is occupancies. A good first model to move.

Estate and practice

Model	What it is	With the SDK
Maven `maven.xcore`	The Maven project model, generated from its schema: modules and their dependencies, as assets with computed ownership and vulnerabilities traced through dependencies.	Loaded through a volume, with the threat facet contributed by the threat floor.
TIBCO BusinessWorks 5 `bw5.xcore`	Process definitions, activities, transitions and groups of an integration product, down to the activity.	Kept as the example of modernizing a legacy system: its processes become executable graphs on Chapter 5, with activities resolved through the capability framework.

Model	What it is	With the SDK
CISO `ciso.xcore`	A security program: incidents, vulnerabilities, awareness campaigns, continuity plans, exercises, third parties and metrics.	On governance and threat. Not reviewed by a security specialist.
System of record `system-of-record.xcore`	An estate of systems of record: datasets, record layouts, copybooks, tables and columns, and mappings to a shared vocabulary.	The vocabulary is a Meta package, and a mapping is a mapping of Chapter 14.

User interface models

Model	What it is	With the SDK
UI `ui.xcore`	A user interface as a model: elements, containers, layout, bindings to values and to events.	To be reviewed against Chapter 6 and Chapter 9. If something remains, it is the stored form of a drawn screen.
Bootstrap UI `bootstrap-ui.xcore`	Bootstrap components on the UI model.	The runtime has HTML and Bootstrap models. Likely retired.
Action, Bootstrap, HTML application	Navigation and pages of generated sites in the older line.	Not migrated. The runtime's site generation and views replace them, if anything is needed at all.
Diagram, presentation, Draw.io	Models for reading and writing diagrams and slides.	**AVAILABLE** in the runtime. Sprotty is not needed. Visio, PowerPoint and Excel wait for models generated from the formats' schemas.

13.4. One user interface or many

The tower's models will have user interfaces in the terminal first, and accounting is the one to start with, because its author is its first user.

The form is not one application for everything. It is a navigator with contributions: the tree of accounts on one side, and on the other a view that depends on what is selected, an account's entries, a transaction, an assertion that a balance is what the bank says it is.

Each view comes from the module that knows the class, as views do everywhere in the runtime. The same form serves the other floors: a tree of capabilities, of controls, of work. It is the navigator with details that Chapter 8 lists as planned.

13.5. Order

The tower moves in the order of dependency, and a floor moves when something needs it.

1. The nano-models of Chapter 11, since everything stands on them.
2. Role, IAM, Seal and Lifecycle as nano-models.
3. Accounting, as the first floor in Meta and the first with a terminal interface.
4. Then by need. The International Banking System decides: what a bank's example requires next is what moves next.

A floor reaches Maven Central when its namespace is settled, it has a book, and something other than itself uses it. Until then its sketch is here.

Chapter 14. OpGraph and NSML

Two designs from the older Nasdanika are to be revisited on the SDK: OpGraph, a model of executable graphs, and NSML, the Nasdanika Semantic Mapping Language. Each has a repository with a design document. This chapter states what each is and what the SDK changes for it, and when it is complete the repositories will be archived with a note that points here, as the Waypoint repository is.

Both are PLANNED, and neither has code in this repository.

14.1. OpGraph

Capability	A process drawn or written as a graph of activities and transitions, which runs, records what it did, and can be replayed on its own diagram.
Status	PLANNED on the SDK. The pre-SDK design is at github.com/Nasdanika-Models/op-graph.
Depends on	Chapter 5 for execution, Chapter 4 for wires, the runtime's Draw.io model for authoring, and the capability framework for what an activity does.
How to help	Bring a process: an integration flow, an approval, an agent's plan. A real one decides what the model has to say.

What it is

OpGraph describes a process as a graph and compiles the graph to a metamodel. Its design-time classes are few: an **activity** is a node, a **transition** is an edge, and a **call** invokes a sub-process, an external system or an agent. A process becomes a class, its activities and transitions become operations, and what an activity calls becomes parameters that are supplied when it runs. Because an activity is an operation with typed parameters, the schema of a tool an agent may call is not written by hand: it is the operation.

A process can be written in a text syntax, drawn in Draw.io, or drafted in a conversation with an assistant. It runs on compiled classes or on dynamic objects, its state can be saved as a checkpoint, in a file or as a commit, and what it did is recorded as telemetry that can be played back on the diagram.

Three uses motivated it. Modernizing a legacy integration product, whose processes are graphs already, is the first. Agentic systems are the second: the same graph, with activities that call a model provider, and with every tool call bounded by a typed operation. The third is orchestration of services in plain Java, where an integration framework would be more than is needed.

What the SDK changes

Most of what OpGraph needed and did not have is now in the runtime or in this roadmap.

- **The carrier.** An execution is a graph of Chapter 5: a fork at a gateway is two commits from one waypoint, a join is a merge, a retry is a sibling, and a checkpoint is a waypoint that was saved.
- **The engine.** The runtime's `Pump` and `Transformer`, where the older design had graph processors. One engine already runs flows, capability resolution and transformations.
- **Two kinds of connection.** A transition carries messages, each of which matters, with Git as the durable log between activities. A wire between properties carries state, of which only the latest matters, and that is a binding. A diagram can say which a connection is.
- **What an activity does.** Resolved through the capability framework, by a selection annotation that names the element of the diagram: the mechanism of Chapter 9, used for behavior.
- **Telemetry.** A span per step, in the runtime's telemetry model, which is kept beside the model it describes. Playing a trace back on its diagram is the Draw.io animation listed in Chapter 10.
- **Settling.** A graph with a cycle stops by a rule over lineage.

So OpGraph on the SDK is expected to be a small model and a thin layer, where the older design had to carry its own runtime.

The order in which to build it is the way of working of Chapter 9: draw a process, run it with nothing connected, then connect one activity at a time. Activities that call a model provider come after inference exists.

14.2. NSML

Capability	A mapping from one model to another, written as rules that can be read, drawn, generated and reviewed, and that record why each element of the result exists.
Status	PLANNED on the SDK. The pre-SDK design is at github.com/Nasdanika-Models/nasdanika-semantic-mapping-language.
Depends on	The runtime's `Transformer` and core evaluators, and Chapter 5 for execution with provenance.
How to help	Bring a mapping you maintain by hand today. Mappings between two versions of a model, and from a system's export to a shared vocabulary, are the plainest cases.

What it is

NSML applies the idea of XSLT to models. A transformation is a set of rules. A rule has a **match**, a class named by its namespace and name, an optional **condition**, and a **produce** clause, a class to create with feature values computed by expressions.

Three decisions shape it.

Types are bound late. A rule names classes by namespace and name, as text, so a mapping can be written by an assistant that has no access to the metamodels, or against a version of a model that does not exist yet. Where the metamodel is available, references are validated against it.

There is no expression language of its own. Every expression says which language it is in. On the SDK that is not a feature to build: the core model has evaluators, for SpEL, XPath, scripts and Groovy, and an expression is one of them.

A mapping compiles to code. Rules compile to a transformer factory, so a mapping written as rules composes with one written in Java.

It has several surfaces: YAML or JSON through the runtime's JSON resources, a Draw.io diagram in which mappings are connections, a text syntax, and a conversation with an assistant that drafts the rules.

Its uses, from the design:

- a model mapped to an application model and from there to a site;
- the context of a step of an executable graph, narrowed to what the step needs, which is also how an agent is given a bounded view;
- migration of a model from one version of its metamodel to the next;
- a view of a federated model with access control applied, so that what a person or an agent may see is a property of the view and not of the consumer's good behavior;
- operations as targets, where a rule produces something that can be called, by a person or an agent;
- filters that shape a resource as it loads, chosen by the name of the file.

What the SDK changes

- **The engine exists.** The runtime's `Transformer` maps a source to a publisher of targets, wires targets to each other as they appear, and detects when nothing can progress. A compiled mapping is a factory for it.
- **Execution is a traversal with waypoints.** Each rule firing is a step whose element is the source object and the rule. Then "why does this target element exist" is answered by walking parents, and it does not have to be reconstructed from logs.
- **A rule that asks a model provider is a rule.** It has the same record, the same telemetry and the same retry as a deterministic one, and the difference is only in its body. A low-confidence answer is retried as a sibling, and the rejected attempt stays for the audit.
- **A mapping is a view over a realm.** Guarded state seen through a mapping is guarded state of another shape. A cheap, deterministic mapping can be a lens, applied on every access. An expensive one, and any that asks a model provider, is applied once and its

result kept, with one complete record of how it was made.

- **Changes and mappings meet.** A change of the change nano-model selects objects and sets values, and a rule matches objects and produces values. Whether a change is a degenerate rule, or a rule produces changes, is to be decided when both exist.
- **The text syntax follows Meta's.** The runtime has a text syntax for Meta with a hand-written grammar and no IDE dependency, and NSML's would be made the same way.

Chapter 15. Technology Models

A technology model describes what a system or a format holds, in the system's own terms: the issues of a tracker, the commits of a repository, the cells of a workbook. It is for reading and writing, and it is tested by round trips. It makes no claim about what the content means. Meaning is given by cross-referencing it with the models of Chapter 13: a person in a Git history is a person in the organization, and a module in a build is an asset in a threat model.

The older Nasdanika has such models, most of them defined in Ecore, each in a repository at github.com/Nasdanika-Models with a loader and a generated site. The ones below are to be migrated. The others stay where they are for now.

15.1. What migration means

A model does not move as it is. Following the rule of Chapter 1, salvage and do not port, it comes as a small model and its integrations come as adapters.

- **The model** is rewritten in Xcore on the runtime's core, generated by Maven with no IDE, with its namespace under the SDK's.
- **The loader goes through a volume.** A system is presented as a volume of models, with relative references, so that a snapshot is a copy, and repeated scans go on until nothing changes.
- **The client is the JDK's.** Where the older loader used a vendor's library, the new one calls the REST API with the JDK's HTTP client, so a model of one system does not bring a second HTTP stack.
- **It gets a book,** with listings from its tests.

A snapshot is the reason to load a system into a model at all. Data loaded from a system is saved and worked on later: enriched in several passes without asking the source again, examined after a temporary token has expired or where the source cannot be reached, and compared with the snapshot of the next quarter. Trackers limit how fast they may be read, and scanning thousands of repositories on every run is slow, so a snapshot is also the practical answer to cost.

Once a system is a volume, Chapter 8 is a way to walk through it, and a file in a repository on a server can be edited where it is.

15.2. The models

Model	What it holds, and its state before the SDK	What the SDK version changes, and what it waits for
Coverage	Java code coverage, loaded from JaCoCo's execution data and class files. Used alone or with the Java model: which methods no test reaches.	Joined with the Java model through references. Waits for the Java model.
Excel	Workbooks, sheets, rows and cells, with a loader and a resource factory, so a workbook is a model resource.	Read through a model generated from the format's schema, with a library only as an optional adapter for formulas and old binary files. Workbooks of the size of a model are the use: configuration, mappings, glossaries, catalogs. Waits for the schema-model route.
Git	The object model of a repository: blobs, trees, commits, references, branches and tags.	A Git volume on JGit, which the runtime's bill of materials has. The core model's Git marker, which records where an element came from, is its first user.
GitHub	Organizations, repositories, people and what they do there. The one with an Xcore sketch, `models/technology/github.xcore`.	A system volume over the REST API, and files committed through it without a clone.
GitLab	Users, groups, projects, repositories, branches and commits, loaded for cross-referencing with an organization: who owns what, which branches are stale.	As GitHub, over the REST API with the JDK's client in place of a client library. The two share the Git model.
Java	Modules, packages, classes, fields and methods, and compilation units, loaded from sources and from the reflection API.	The runtime has a JVM model, of a running program. The Java model is of sources: for analysis, for generation, and with coverage for finding what to test.

Model	What it holds, and its state before the SDK	What the SDK version changes, and what it waits for
Jira	Users, projects, issues of several types and the relations between them, categories, releases, filters and boards.	A system volume with snapshots. Comparing the state of a hierarchy of projects once a quarter is its plainest use.
SQL	Two models: the metadata of a database as JDBC reports it, with generators of models from it and loading of data into them; and the syntax tree of SQL statements.	Metadata first, as a system volume. The syntax tree follows the pattern of the Cypher model.
Python	A model for generating Python sources.	Perhaps not migrated. Nothing on this roadmap generates Python.

None of these has a date. Each moves when something needs it, and the order of the table is alphabetical, not a ranking. The candidates to go first are Git, since provenance already refers to it, and whichever of GitHub and GitLab a first reader asks for.

15.3. Formats already in the runtime

For completeness: the runtime 2026.11.0 has models of Draw.io diagrams, of diagrams in general with layout by ELK, of SVG, of presentations, of HTML with Bootstrap, and of TypeScript and React. They are **AVAILABLE** and described in their own books. Visio, PowerPoint and Word wait, with Excel, for models generated from the formats' schemas.

Appendix A: Items

Every item of the roadmap, with its status and where it is described. An item has an id that does not change when its title does, so that another plan can refer to it.

SKETCH means there is code in this repository, built and tested with every build. PLANNED means designed or intended, and not built.

A.1. Runtime

Id	Item	Status	Where	Depends on
runtime.binding	Bindings	SKETCH	Chapter 4, runtime/common	Runtime common
runtime.binding.list	List bindings with changes	PLANNED	Chapter 4	runtime.binding
runtime.waypoint	Waypoints, state, realms, telemetry	SKETCH	Chapter 5, runtime/common	Runtime common
runtime.waypoint.git	Waypoints in Git, and escalation	PLANNED	Chapter 5	runtime.waypoint, a Git volume
runtime.widget	Widgets, kinds and Markdown	SKETCH	Chapter 6, runtime/widget	runtime.binding
runtime.widget.asciidoc	An AsciiDoc context with files split by policy	PLANNED	Chapter 6	runtime.widget
runtime.widget.html	An HTML context with Bootstrap	PLANNED	Chapter 6	runtime.widget
runtime.widget.interaction	Actions, selection, input, tree and form	PLANNED	Chapter 6	runtime.widget
runtime.tui	The terminal on TamboUI, and screenshots	SKETCH	Chapter 7, runtime/tui	runtime.widget
runtime.tui.kit	A base terminal command and the kit	PLANNED	Chapter 7	runtime.tui

Id	Item	Status	Where	Depends on
`runtime.tui.telemetry`	Telemetry for commands in the terminal	PLANNED	Chapter 7	Runtime `cli`
`runtime.commander`	Nasdanika Commander	SKETCH	Chapter 8, `runtime/commander`	`runtime.tui`, runtime `volume`
`runtime.commander.tree`	Commander on any tree, with commands by type	PLANNED	Chapter 8	`runtime.commander`
`runtime.screens`	Screens drawn in Draw.io	PLANNED	Chapter 9	`runtime.tui`, runtime `drawio`
`runtime.inference`	Inference and provider modules	PLANNED	Chapter 10, Chapter 12	`model.tool`
`runtime.mcp`	MCP on a JSON-RPC core	PLANNED	Chapter 10	`model.tool`
`runtime.telemetry.live`	Live telemetry	PLANNED	Chapter 10	Runtime `common`
`runtime.telemetry.sanitizer`	A sanitizer for telemetry	PLANNED	Chapter 10	Runtime `common`
`runtime.volume.system`	System volumes	PLANNED	Chapter 10	Runtime `volume`
`runtime.autosave`	Model auto-save on change recording	PLANNED	Chapter 10	Runtime `cli`
`runtime.graph`	Reactive compute graphs	PLANNED	Chapter 10	`runtime.binding`, `runtime.waypoint`
`runtime.cli.commands`	Commands over what is built	PLANNED	Chapter 10	Runtime `cli`
`runtime.cli.install`	`nsdk install` and profiles	PLANNED	Chapter 10	Runtime `bootstrap`
`runtime.doc.reference`	Generated metamodel references	PLANNED	Chapter 10	Runtime models
`runtime.merger`	A merger for generated files	PLANNED	Chapter 10	Runtime `volume`
`runtime.ui.browser`	The polymorphic user interface in a browser	PLANNED	Chapter 10	`runtime.widget.html`

A.2. Models

Id	Item	Status	Where	Depends on
model.core.related	Related	SKETCH	Chapter 11, models/core	Runtime core model
model.core.change	Change	SKETCH	Chapter 11, models/core	Runtime core model
model.core.occupancy	Occupancy	SKETCH	Chapter 11, models/core	model.core.related, model.core.change
model.core.kind	Kind	SKETCH	Chapter 11, models/core	model.core.change, model.core.occupancy
model.core.enterprise	Enterprise	SKETCH	Chapter 11, models/core	Runtime core model
model.core.document	Document	SKETCH	Chapter 11, models/core	model.core.enterprise
model.core.example	Example	SKETCH	Chapter 11, models/core	Runtime core model
model.core.role	Role, IAM, Seal and Lifecycle as nano-models	PLANNED	Chapter 11, models/core/sketches	model.core.occupancy, model.core.kind
model.core.change.engine	Applying changes	PLANNED	Chapter 11	model.core.change
model.cli	The CLI model and self-documenting command lines	PLANNED	Chapter 12, models/runtime	Meta, runtime.widget
model.tool	The tool model	PLANNED	Chapter 12, models/runtime	Meta, model.core.example
model.inference	The inference model	PLANNED	Chapter 12, models/runtime	model.tool
model.expression	The expression model	PLANNED	Chapter 12, models/runtime	Runtime core model
model.cypher	The Cypher model, parser and engine	PLANNED	Chapter 12, models/runtime	model.expression
model.xpath	XPath for models	PLANNED	Chapter 12	Runtime core model

Id	Item	Status	Where	Depends on
`model.tower`	The model tower on the SDK	PLANNED	Chapter 13, `models/tower`	`model.core.role`, Meta
`model.tower.assembler`	Products on projections of the tower	PLANNED	Chapter 13	`model.tower`, `runtime.ui.browser`
`model.opgraph`	OpGraph on the SDK	PLANNED	Chapter 14	`runtime.waypoint`, `runtime.screens`
`model.nsml`	NSML on the SDK	PLANNED	Chapter 14	`runtime.waypoint`
`model.technology`	Technology models: coverage, Excel, Git, GitHub, GitLab, Java, Jira, SQL	PLANNED	Chapter 15, `models/technology`	`runtime.volume.system`

The models in `models/runtime`, `models/tower` and `models/technology` are marked PLANNED although each has an Xcore file there. The files are the pre-SDK models, kept to be read. An item becomes a SKETCH when it is rewritten on the SDK and builds here.

Appendix B: Revisions

The chapters say what holds now. This appendix says when it came to hold, newest first. An entry has a date, what was decided, and the chapters it changed. Nothing in a chapter changes without an entry here, and an entry is never rewritten: a later decision gets a later entry.

The entries before 2026-11-11 summarize decisions made while the roadmap was a private file of dated sections. They are here so that the first cut of the book has its history, and only what is public in them is carried over.

Date	Decision	Chapters
2026-11-11	The roadmap becomes a repository in the SDK organization: this book, modules built on runtime 2026.11.0, and model sketches. Bindings, waypoints, widgets, the terminal and the commander get code. The Maven group is `org.nasdanika.sdk.roadmap`, and a module graduates by dropping `roadmap` from its names.	All
2026-11-10	Bindings are a supplier and a publisher that emits the current value on subscription, closeable, in `common` beside `Flows`, and are not specific to widgets. "Resolve" or "select" is the word for finding what implements a diagram element, and "binding" is kept for values.	Chapter 4, Chapter 9
2026-11-10	Values reach telemetry through a sanitizer, later: as they are, masked, encrypted or tokenized.	Chapter 10
2026-11-10	Bindings between processors make reactive compute graphs, which settle by lineage. Waypoint becomes part of the runtime: its core without a reactive library, with telemetry folded in, and with Git, EMF Transaction and the mapping language outside.	Chapter 5, Chapter 10
2026-11-10	The public roadmap becomes a book in AsciiDoc with chapters by area and dated revisions. Roadmaps and architecture repositories may depend on each other as builds do.	Preface, Chapter 10

Date	Decision	Chapters
2026-11-09	Widgets: a `widget` module with `Widget<R, C>` and kinds asked for by context type, not by instance. The first kinds are text, link, section, lists and table. Diagnostics go through the context. Markdown is in the module, HTML with Bootstrap beside the HTML models, and TamboUI in `tui`.	Chapter 6
2026-11-09	AsciiDoc is a format of several files: sections are derived contexts, and the context decides by a policy which become files, by level first. Links refer to ids. The first consumer of widgets is the documentation of Meta.	Chapter 6
2026-11-09	Terminal widgets are written against `TerminalContext`, with TamboUI as its provider, and `TerminalContext` extends `SurfaceContext`, which a Draw.io context can share. Terminal screenshots are TamboUI's own SVG export.	Chapter 6, Chapter 7
2026-11-09	A command tree is saved as a model and rendered by widgets, for any picocli application. Model auto-save is built on change recording.	Chapter 12, Chapter 10
2026-11-05	A terminal user interface on TamboUI, in a `tui` module, with `cli` free of it. For technical people the terminal comes before the browser.	Chapter 7
2026-11-05	Nasdanika Commander: two panels over volumes, in a `commander` module, with the classic keys and a key bar, and with move, new directory, directory copy and delete, and an overwrite prompt in its first version. Later: any tree, commands by type, a navigator with details, extensions.	Chapter 8
2026-11-05	Logical colors are the only colors a terminal view names, mapped to the terminal's palette.	Chapter 6, Chapter 7
2026-11-05	Screens drawn in Draw.io, after the release: draw, run with nothing bound, then bind. One-way mapping through the capability framework with selection annotations, two shape libraries, flows that connect screens, and documentation from the drawing.	Chapter 9
2026-11-05	The runtime release is 2026.11.0. An installation is extended only through the mirrors an organization has configured.	Chapter 1, Chapter 10

Date	Decision	Chapters
2026-10-30	Meta has a text syntax. A request for a view is a point in coordinates: the class and kind of the element, the kind of representation element, the format, the channel.	Chapter 11
2026-10-29	The enterprise model becomes a nano-model in core: stakeholders, concerns, viewpoints, formats and channels. A document is a view and a section a representation.	Chapter 11
2026-10-28	A view covers the features it shows and passes the rest to less specific views.	Chapter 10
2026-10-27	Content digests as identifiers of model elements. Nasdanika Field Notes for what is learned about third-party technologies. The look of the books and the NBOK appendix are artifacts of the runtime that other repositories unpack.	Chapter 10, Appendix C
2026-10-25	The plan is compacted: decisions that hold, a queue ranked by value for effort, and dated sections after it. The reflective viewer becomes the polymorphic user interface. Draw.io is rendered by its own viewer.	Chapter 1

Earlier decisions, from September and October 2026, shaped the runtime that is released, and the SDK book argues them: one identity per module, the foundation of Java, EMF core and the OpenTelemetry API, flows on the JDK, volumes, telemetry per unit of work, and the three planes.

Appendix C: Building and Conventions

C.1. The repository

```
pom.xml                    the parent: the runtime's bill of materials, versions, the modules
AGENTS.md                  how the repository is laid out, for AI assistants
runtime/                   what is on the roadmap for the runtime, as code
  common/                  bindings and waypoints
  widget/                  widgets, kinds, Markdown and the neutral terminal context
  tui/                     the terminal on TamboUI, and screenshots
  commander/               Nasdanika Commander
models/
  core/                    the core nano-models: Xcore that generates and compiles
    sketches/              Role, IAM, Seal and Lifecycle, as they are before the SDK
  runtime/ tower/ technology/   pre-SDK and drafted models, Xcore to read
tests/
  widgets/                 one view rendered four ways: tests across modules
book/                      this book, and the site built from it
.github/workflows/         the build that publishes the site
```

A module has the layout of a runtime module: `pom.xml`, `src/main/java` with a `module-info.java`, and `src/test/java`. Its group, module name and packages are the runtime's with `roadmap` added, and it depends on the runtime's modules through the runtime's bill of materials, with no versions of its own.

C.2. Building

Java 25 and Maven 3.9 are needed. From the root:

```
mvn verify                                everything: modules, tests, models, the book
mvn verify -Dasciidoctor.skip=true        everything but the book
mvn verify -Dbook.editor-notes=false      the book without editor's notes, as published
mvn -pl runtime/commander -am package     one module and what it needs
```

The runtime comes from Maven Central, at the version of the `nasdanika.runtime.version` property. To try the roadmap against a runtime that is not released, install that runtime locally and pass its version:

```
mvn verify -Dnasdanika.runtime.version=<version>
```

Nothing here is deployed to Maven Central.

C.3. The book

The book is AsciiDoc in `book/src/docs/asciidoc`, rendered by Asciidoctor to a PDF, an EPUB and one HTML page.

Its build does five things, in order.

1. Unpacks the look of the HTML and the NBOK appendix from the runtime release, the artifacts `org.nasdanika.sdk.runtime.doc:zephyr` and `:nbok`.
2. Copies the manuscript to `target/manuscript`, and beside it what the tests wrote: the screenshots of the commander and the renderings of Chapter 6.
3. Draws the cover, with `book/src/cover/Cover.java`, run by the JDK that runs Maven.
4. Renders the three formats into `target/book`.
5. Assembles `target/site`: the HTML as `index.html`, and the PDF and the EPUB under names that do not change, so that links to them keep working.

The modules are built and tested before the book, because the book includes what their tests wrote. A warning from Asciidoctor fails the build: a missing tag, a broken cross-reference, a missing image.

The PDF uses the theme of the SDK book, and the HTML the Bootswatch Zephyr look of the Nasdanika Zephyr Books.

C.4. Publishing

A workflow in `.github/workflows` builds the repository on every push and pull request. On a push to `main` it also publishes `book/target/site` to GitHub Pages, which serves it at roadmap.sdk.nasdanika.org. The book on Leanpub is uploaded by hand, the PDF and the EPUB from `book/target/book`, when a release of the runtime is made or the roadmap has changed enough to say so.

C.5. Conventions

Listings are includes. Code in a chapter comes from a tagged region of a source file or a test:

```
include::{flow-test}/BindingTests.java[tag=first,indent=0]
```

A callout is a numbered comment at the end of a line of the source. Code that is not compiled anywhere, such as a design for something not built, is shown as such and said to be so.

A test is named for what it shows. `aSlowSubscriberGetsTheLatestValueNotABacklog` is a sentence of the chapter it is quoted in.

Status labels, SKETCH and PLANNED, on anything that is not in a released runtime. The preface says what each means.

Editor's notes record what is known to need a decision or a check. They are `IMPORTANT` blocks with the caption "Editor's note", inside a condition on the `editor-notes` attribute, shown by default and left out of published editions.

Sources are stated. Where a statement comes from running something, the test is there. Where it comes from reading code, or from memory, the text says so.

Voice. First person, the author's. No long dashes in prose.

C.6. Adding an item

1. Add a row to Appendix A, with an id that will not change.
2. Write the chapter, or the section of one, with the six things an item says: Chapter 2 lists them.
3. If there is code, add a module under `runtime/` or a model under `models/`, with tests, and tag what the chapter quotes.
4. Add an entry to Appendix B if a decision was made or changed.

C.7. When an item graduates

The module moves to the runtime's repository, and `roadmap` is dropped from its group, its module name and its packages. Its tests become tests of the runtime, and its chapter becomes a chapter of the runtime's books, with its listings following the tests. Here, its row in Appendix A is removed, its chapter is replaced by a line in Chapter 1 that says where it went, and Appendix B records the release it went into.

C.8. TamboUI on the module path

TamboUI 0.7.0 keeps its exporters in packages that its core module does not export. The `tui` module's screenshots use them, so wherever that code runs on the module path, three options are needed. The root `pom.xml` has them as the `tamboui.exports` property:

```
--add-exports=dev.tamboui.core/dev.tamboui.export=org.nasdanika.sdk.roadmap.runtime.tui
--add-exports=dev.tamboui.core/dev.tamboui.export.svg=org.nasdanika.sdk.roadmap.runtime.tui
--add-exports=dev.tamboui.core/dev.tamboui.export.text=org.nasdanika.sdk.roadmap.runtime.tui
```

The commander's jar runs on the class path, where they are not needed.

Appendix D: Glossary

Binding

A value that can be read now and followed as it changes: a supplier, a publisher that starts with the current value, and closeable. Not to be confused with resolving a diagram element to what implements it, or with a decision binding of the model tower.

Catalog

Definitions published together as a resource that other models refer to: kinds, changes, dimensions, examples, stakeholders.

Channel

A way of delivering a view: e-mail, a site, a repository, MCP, print. A channel accepts formats.

Change

A change to a model as data: a selector that finds what to change, and a value, possibly computed. A plan that can be reviewed before it is applied.

Coalescing

Keeping only the latest value for a subscriber that is slower than its source, where a stream would hold the source back.

Context

Where a widget renders: Markdown being written, an area of a terminal. Its type is the format. A derived context is one made from another for a part of the output: a deeper heading level, a sub-area, a buffer.

Facet

What a waypoint carries besides its lineage, asked for by type.

Floor

A model of the model tower, which builds on the models below it.

Format

A way of writing a representation down: Markdown, AsciiDoc, HTML, SVG.

Gradient ascent

How the next step is chosen: from where things stand, the step with the most value for its time and effort, counting what it makes cheaper later.

Graduation

The move of a module from this repository to the runtime, when its identity is settled, it is documented, and something other than itself uses it.

Item

A thing on the roadmap, with an id, a status, what it depends on and how to help.

Kind (model)

A classification of an element as data, which may carry changes that configure what it classifies.

Kind (widget)

A capability interface a view is written against, such as `Table`, provided per type of context.

Lineage

The waypoints an execution passed through to reach a waypoint.

Micro-model

A small model in a module of its own.

Nano-model

A small aspect in one Xcore file, in a package of the core module.

NBOK

The Nasdanika Body of Knowledge: the books, stories and example code about Nasdanika. This book is part of it.

North Star

Agentic work on bounded, typed models, with state in Git and the safety of a transaction.

Occupancy

The recorded fact that a subject is in a region: a box in dimensions, of which time is one.

Polymorphic user interface

A user interface assembled from views chosen by the class of what they show and by the format asked for.

Provider

An implementation behind an interface that views are written against: TamboUI is a provider of the terminal context.

Pump

The runtime's queue of work drained by one thread at a time. The realm of bindings and of a terminal's frame loop.

Realm

A space of state that is reached only through commands, so that its rules hold.

Resolve

To find what implements an element of a diagram. Kept apart from "bind", which is for values.

Sketch

Code in this repository, built and tested, and not settled.

Surface

A context that is an area: a terminal in cells, a diagram page in pixels.

Technology model

A model of what a system or a format holds, in its own terms, for reading and writing.

Viewpoint

A way of looking that frames concerns. A view is what is built from a viewpoint for some elements.

Volume

The runtime's file abstraction: a tree of entries with content, wherever it is. A panel of the commander is a volume.

Waypoint

Where an execution is and how it got there. A value, like a commit: advancing makes a new one.

Widget

Something that renders itself into a context. A value, built before any context exists.

Wire

A connection drawn between two elements, where one follows the other's value.

Appendix E: The Nasdanika Body of Knowledge

This book is part of the Nasdanika Body of Knowledge, NBOK for short: the books, stories and example code about Nasdanika, organized so that people and AI assistants can find what they need in it and rely on what they find. Every book in it ends with this appendix, word for word, so that whichever one you are reading, you know where the others are and how they fit together.

Bodies of knowledge such as SWEBOK for software engineering and PMBOK for project management are written guides. NBOK is built. Its listings are regions of code that is compiled and tested before a book is rendered, so a book whose code no longer does what it says does not build.

E.1. What it holds

- **The SDK book**, *Nasdanika SDK Runtime: More Structure with Less Effort*: the overview, with the why, the key capabilities and the high-level how. Read it first.
- **Nasdanika Zephyr Books**: one per module, or per small group of modules used together, each the detail for its modules, as user guide, developer guide and reference. Each stands on its own.
- **Readmes**: one page on what a module is and why. A module with a Zephyr Book may have none.
- **Nasdanika SDK Stories** PLANNED: one use case each, related to the Nasdanika SDK in some shape or form, written as a Maven module whose code, if it has any, is the source of its listings, and collected each year.
- **The International Banking System** PLANNED: the practice of architecture as code, taught as a course on one example that grows with the platform. Inspired by the C4 Model "Internet Banking System".
- **Nasdanika Field Notes**: short articles about the technologies and services Nasdanika is built and published with, such as EMF, each verified by a test where it can be, and recipes, such as publishing a book, collected in one book. Their tests use the third-party versions of the runtime's BOM, so every runtime release checks the runtime's

assumptions about its dependencies. Unlike the rest of NBOK, they are about other projects' code, and useful without Nasdanika.

A story is à la carte; the International Banking System is the course; a field note is about the ground they stand on.

E.2. How it is laid out

Every repository of NBOK follows one layout, the Maven layout with a few additions:

```
pom.xml                          the reactor: its modules, their names and descriptions
doc/                             documentation about the repository as a whole
  book/manuscript/               a book about the repository, if it has one
  book/examples/<topic>/         that book's examples, one Maven module per topic
<module>/
  pom.xml                        the module's name and description
  readme.md                      what the module is and why, on one page
  src/main/java/                 the code, with module-info.java
  src/main/resources/            resources, and Xcore models under the package path
  src/test/java/                 the tests: the source of the book's listings
  src/test/resources/            test data: the source of the book's figures
  src/docs/asciidoc/index.adoc   the module's Zephyr Book, with front/, chapters/
                                 and appendices/ beside it
```

A module has a Zephyr Book when it has `src/docs/asciidoc/index.adoc`. A story PLANNED is a module at `<year>/<topic>/` in the `stories` repository, with `story.adoc` and `index.adoc` in `src/docs/asciidoc/` beside its code, and `<year>/book/` builds the year's collection from the stories next to it. A field note is a module at `<technology>/<topic>/` in the `field-notes` repository, with `article.adoc` and `index.adoc` in `src/docs/asciidoc/` beside its tests, and `book/` builds the collection.

E.3. How it is published

- **Maven Central**, per module and version: the jar, its `sources` and `javadoc`, and, for a module with a Zephyr Book, the book as PDF and HTML with the classifier `book` and the tests with the classifier `test-sources`, the complete source of every listing. Central is permanent, so the book of every released version stays available.
- **The runtime's site**, PLANNED runtime.sdk.nasdanika.org, with the latest books.

- **Leanpub**: the SDK book, with the Zephyr Books that have readers beyond the runtime in a bundle, and the yearly collection of stories PLANNED, each free with a suggested price.
- **Zenodo**, sources published by GitHub and books uploaded manually.
- **Lulu** for print-on-demand copies of the books.

E.4. How to read it

People read the PDF or the HTML: the SDK book for the why and the big picture, then the Zephyr Book of the module at hand.

AI assistants are better served by the sources than by the renders. AsciiDoc is a fraction of the size of the HTML, whose markup and highlighting multiply it, and the sources lead to the code itself. In a repository of NBOK, an assistant finds its way as an engineer does:

1. The root `pom.xml` lists the modules, with a name and a description each.
2. A module's `readme.md`, where it has one, says what the module is.
3. A module's `src/docs/asciidoc/index.adoc` includes the book's files, and each listing's `include::` names the test file and the tagged region it comes from: follow it to the code.
4. A term is in a book's glossary appendix, and a convention in its rules or building appendix.

Two aids for assistants are planned PLANNED: each book and story also as one AsciiDoc file with its includes resolved, so that it is complete without the repository, and an index generated from the headers of the books and stories (title, description, keywords, version), so that an assistant can choose the few it needs instead of reading all of them.

E.5. Why it can be trusted, and for which version

- **Listings are code.** Every listing is a tagged region of a test or an example that is compiled and run before the book is rendered, and a warning, such as a missing tag, fails the build.
- **Every book has a version.** A book is built with a release of the runtime and carries its version, here 2026.11.0. It is true for that version; for another, read the book of that

version.

- **Status labels** mark what is not built yet: SKETCH for what is designed and partly built, PLANNED for what is designed and not built. Everything without a label is available.
- **Editor's notes**, the author's notes on what is still to be checked, are left out of published editions.
- **Stories are built twice** PLANNED: against the release each was published with, so that it reproduces, and against the next one, so that a story that no longer applies is marked before a reader finds out.

The source of this appendix is `doc/nbok/nbok-appendix.adoc` in github.com/Nasdanika-SDK/runtime, published with each release as `org.nasdanika.sdk.runtime.doc:nbok`.

About the author



I am Pavel Vlasov, the creator of Nasdanika and of the roadmap this book describes. I have worked in information technology for more than thirty years, thirty of them in financial services, as a developer, an architect and an enterprise architect.

I hold a master's degree in plasma physics from the Novosibirsk State University. I am certified as a Java developer and as a Java architect, and as a TOGAF Enterprise Architecture Practitioner.

I write about models, architecture and AI-assisted development at medium.com/nasdanika. My code is on GitHub, and I am on LinkedIn.