The Ocarina Holy Book

Igor Casanova

Table of content

Chapter 1 What is Ocarina? 1

- Another one! 1
  - Them 1
  - Us 2
- The cultural problem 3
  - ISTQB, where are you? 3
  - Back to basics 4
- The real pain point 4
  - About "creativity" 4
  - What's the science? 5
- Vision 5
  - Sovereign grammar 5
  - Born from the field 6
  - Adoption 6

Chapter 2 First feedbacks 7

- First grievances 7
- Performative complexity 8
- Philosophy 8
  - KISS 8
  - What's being "disruptive" 9
  - Incorruptible 10
- First relevant interaction 11
  - The TRUE solidarity 11
  - Why not Playwright? 11
- Secession 11
  - Anti No-Code 12
  - Anti devs 13
  - Anti startup nations 13
  - Anti hype 13
  - Anti slipologists 14

Chapter 3 First steps 16

- Disclaimer 16
- 1. Project setup 16
- 2. Adapters 16
  - 2.1 EnvGetters 16
  - 2.2 Act 18
  - 2.3 TestCampaign 18
  - 2.4 TestSuite 19

2.5 MatchPage . . . 20
3. Writing a first POM . . . 20
3.1 SeleniumTitleMixin . . . 21
3.2 Returning self . . . 21
4. Writing connectors . . . 21
5. Writing a first test scenario . . . 22
6. Creating a test suite . . . 23
7. Creating a test campaign . . . 23
8. Creating a test cycle . . . 23
9. Bootstrapping the project . . . 24
Chapter 4 First scenarios . . . 26
The end of invalid states . . . 26
Act and drive_page . . . 26
match_page . . . 29
Repetitions . . . 30
Fragments . . . 31
Aliasing . . . 32
Chapter 5 First jutsus . . . 34
Datasets . . . 34
Smoke tests . . . 36
Setup and teardown . . . 36
Lifecycle . . . 36
Proxy pattern . . . 37
Reactive programming: NO . . . 38
Architectural answer . . . 38
API calls and locks . . . 39
Browser profile . . . 39
Chapter 6 First real-world hurdles . . . 41
Random server errors . . . 41
Random errors within a step . . . 42
Selenium random errors . . . 44
Discrete random errors . . . 44
Using Javascript . . . 46
Fingerprinting . . . 46
Report . . . 46
HumanizedDriver . . . 46
Concurrency Heisenbugs . . . 46
Chapter 7 Extensibility . . . 48
ValidationChain . . . 48
Chaining invariants . . . 48

Chaining validations 48
Reusable invariants 48
Custom assertions 49
Type safety 49
Success and failure 49
Plugins 51
Extensible grammar 51
Chapter 8 Using Ocarina with AI 52
The three spiritual stones 52
CLAUDE.md 52
skills/ 52
Review (13) 52
Analyse (4) 53
Black-hat (6) 53
Comprehend (4) 53
Pick (3) 53
Author (8) 53
Refactor (2) 54
State (1) 54
Setup (1) 54
Run (1) 54
Recurring chains 54
Discipline 54
What this setup isn't 55
Exposed resources 55

Chapter 1

What is Ocarina?

Every testing framework made the same wrong bet. Ocarina doesn't. Find out why.

Ocarina was designed to make **automated browser-based tests as simple as possible**, while giving its user **total control**.

Another one!

Them

Most testing frameworks were built in a world where the barrier between "those who code" and "those who define tests" was real and structural.

Robot Framework tried to **work around it** with a *DSL* (*Domain Specific Language*), complete with **its own format** and **its own plugin ecosystem**. In doing so, RF imposes its own standards by default: that's the immediate cost of its promise.

Cucumber tried the same with *Gherkin*: a "natural" language that, in practice, **constrains everyone without truly freeing anyone**. Cost: **a permanent translation layer, Gherkin/code desynchronization**.

All of them bet on the same thing: **hiding complexity** to bridge the gap between profiles.
Result: non-technical people remain spectators, and technical people end up **trapped in a tool they would never have chosen**.

The biggest cost is a **race to the bottom: fewer options, and a "flexibility" that can only be achieved by *fighting* your *tools* rather than using *solutions***.

There's no use pulling on the rope we've each been given if what lies at the center is just a Gordian Knot.

Us

Better alone than in bad company.

Ocarina bets on the opposite: this barrier will disappear. This is a non-debate around which everyone has tied a noose with needlessly complicated tools, mistaking them for solutions. Impact: **operational disaster** the moment a need arises that can't be expressed within a "framework" that is NOT truly generic.

And the worst part: **all these technologies will keep evolving in the same direction**. NONE of them will make this *shift*, because it would require a paradigm change, **a return to fundamentals that directly contradicts their entire value proposition**.

Yet what remains is the need for test code that is **readable, traceable and flexible** in its most **raw** form.

With AI, and tools like *Claude Code*, this bet grows stronger every day.
The bridge between technical and non-technical people is no longer an abstraction layer.

It's AI itself. AI that works on raw data.

The cultural problem

There is another blind spot, rarely named: **methodology**.

ISTQB, where are you?

The ISTQB and professional testers have spent decades building a precise, battle-tested vocabulary: *test cycles*, *campaigns*, *test suites*, *test cases*, *test steps*. A clear hierarchy, designed to **organize, trace and drive** software quality.

Automated tools have **largely ignored this legacy**.

pytest, *Jest*, *Mocha*... all are **hybrid mixes** where testers must learn to think like developers, and where nobody truly speaks the same language.

This "two-language method" is a failure.

Back to basics

Ocarina makes no such compromise.

Its structure is modeled **directly and exclusively** on the tester's methodology. Every concept in the code maps to a domain concept in testing. No borrowing, no repurposing, no *"close enough."*

And because Ocarina commits to this bet all the way: **it is entirely autonomous**. No pytest plugin. No forced integration into a third-party ecosystem. Ocarina is *batteries-included*, it needs nothing else to operate, and that is a deliberate choice.

The real pain point

About "creativity"

Everyone obsesses over the *how*: interfaces, abstractions, "pretty" DSLs.
Meanwhile, the *why* disappears.

Ocarina makes the opposite choice.

Its entire design and this Holy Book are focused on the **why**: on real problems.
Not on an abstraction to use *"however you like."*

As for the *how*?
The answer is simple: Ocarina is **dense, immediately operational and strict**. Built so that both humans and LLMs can understand its core and usage without friction.

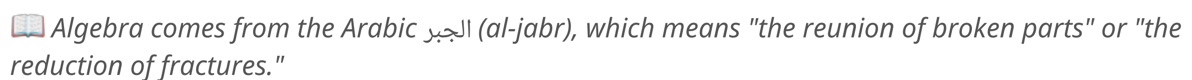
What's the science?

Here, everything rests on **static foundations**:

- types,
- generics,
- functional programming.

Ocarina makes its misusing difficult by design: the compiler shall prevail.

Just algebra.



> 📖 *Algebra comes from the Arabic الجبر (al-jabr), which means "the reunion of broken parts" or "the reduction of fractures."*

Most tools start from human grammar to supposedly "formalize."
Then comes the realization that a machine can't read it as-is.
So, let's pile on "adapters."

We don't call that *formalization* but *wishful thinking*.

Ocarina obviously does the **opposite**.
That's what makes Ocarina **stable and extensible at the same time**.

Vision

Sovereign grammar

What if delegating your grammar to "standards" was never a good idea in the first place?

The real gap with E2E testing **is not imposing the "best" Newspeak**.
Short answer: Ocarina is **extensible**. Any verb and conjunction can be created, all governed by **strict rules that keep the whole thing deeply consistent**.

What remains is to guarantee **traceability and robustness**.

Born from the field

In Ocarina, every step is observable.
The error path is explicit. The test report **emerges naturally from the code**.

No over-engineering. No unnecessary dependencies.
Just something **small, readable, and built to last**.

The most powerful thing is that **Ocarina invents nothing: Ocarina returns to fundamentals**.

Adoption

In the most extreme cases: Ocarina doesn't need to be installed.
Copy it, adapt it, run it.
No dependency to audit.

For teams constrained by *security policies*: the code is small, auditable in an afternoon. Nothing hidden. The only external dependencies live in the post-execution plugins and if one of them doesn't fit, it can be removed without breaking anything else.

In practice, a consultant can show up at a client site with Ocarina in their pocket, *almost* without asking anyone's permission.

People don't grow weary, outdated strings wear them down.

And that is precisely why it exists: to give testers back their **independence**.
All of this, in **a jewel of synthesis**.

Chapter 2

First feedbacks

The first criticism made of Ocarina. Guess which one.

From the start, Ocarina was quietly introduced to professionals across the field, from all backgrounds.

First grievances

The first "criticisms" were often the same: *"You're bragging about something that's very simple."*

Some stopped there.
Others went further: trying to explain what "real complexity" is from their opinion.

But **none of them would have been able to achieve the same thing.**

The funniest point is that **I didn't even have to bring up complexity myself.** I just **presented some test scenarios written with Ocarina**, and **that reaction came up immediately**.

It's just the result of the operant conditioning of 99% of the folks in our industry.

Adding yet another cog to an incomprehensible machine seems to be the fetish of many Mojo engineers.

For those, opening the hood was enough.

First things first: this is not about bragging.
Ocarina **is not the product of showmanship**, unlike what many "engineers" tend to produce. It is simply the product of what *emerges naturally*.

Narcissism expresses itself through making everything shamefully complicated for the sole purpose of *domination*. Not through acquiring skills and simplifying processes, since the end result is total malfunction: chaos.

This is also one of the ways you can quickly spot narcissists in a *casino*: the more a machine flashes, the more "complicated" it seems, the more a narcissist will want to sit down and prove they are "smarter than the machine."

They can also be recognized by their *intolerance of failure*. They systematically want to show how much THEY have "thought of everything."

Never any red with a narcissist, only green, only "ready to ship to production."

The consequences are dramatic, on top of being radically opposed to the mindset of software testing: narcissism brings nothing good in this domain.

In any case, it brings absolutely nothing good anywhere: it brings nothing but *hell on earth*.

Ocarina is radically opposed to these phenomena.

Performative complexity

Real complexity doesn't show off. It is felt. In stability, in extensibility, in what doesn't break.

There is no added value in building something horrendously complicated to use, other than proving that it's complicated. Nobody asks for that.

Our industry has a serious problem with complexity being treated as a badge of seriousness.

This complexity **exists solely to "impress peers,"** and achieves nothing but **outdoing oneself in stupidity**.
Excelling in the wrong direction is worse than plain mediocrity and that is the only "achievement" to see in it: congratulations, you have successfully reached the top of the **pyramid of bullshit** where no self-questioning exists anymore!

This is also why **KISS** (*Keep it simple, stupid*) is so misunderstood in the industry. Many assume "simple" means "unsophisticated." It would be hard to misread a principle more thoroughly.
So we end up with the worst of both worlds.

How can one claim to be *productive* in this way? Seriously?

You. Don't. Even. Know. What. "Clean code". Actually. IS.
You didn't even try it!

Philosophy

KISS

KISS is at the heart of Ocarina.

And if, upon reading a first test scenario, the reaction is *"this is super simple..."* it would be a shame to mean it as criticism: that is exactly the compliment being sought.
Thank you.

To get there, the question was never about "shining" brighter than anyone else.
That was never really a challenge to begin with.
The question was simply: **what do I actually need?**

Some folks had other ideas, quite "creative" ones at that:

- Twisting Ocarina's ROP (*Railway Oriented Programming*) implementation until it lost all meaning, since they didn't even know what ROP is,
- Dropping "*hooks*" and other so-called "*ninja techniques*" right in the middle of test steps,
- Abandon typing,
- Rewrite in Rust for "performance" concerns,
- Forcing their clueless take on lazy evaluation and *IoC* down my throat like it's gospel,
- "Explaining" imperative vs. declarative programming to me while spewing complete nonsense,
- "Talking" to me about *event-driven programming*, while still spouting nonsense,
- And worst of all, pontificating about a horrifying theory of "declarative object-oriented programming,"
- Etc.

Even someone I had a great deal of respect for until then started getting on my nerves.
He thought he had things to teach me, acting arrogant, telling me "what was missing" in a disagreeable tone, when everything he was talking about was not only on the roadmap, albeit still deliberately hidden for strategic reasons, but would go far beyond anything he could ever imagine.

Me, I said nothing, I stayed silent after sending the *repo*.
Him, immediately, he "knew everything" better than everyone else.

What's being "disruptive"

> *Well, right now I'm doing YC and my client is IBM.*

> *Yeah and I know the Queen of England.*

> *Have I ever told you about the yakuzas?*

> *I went to a top engineering school and since then I've learned nothing new.*

Sure: let me quote the motto of the first Indonesian hackers crew I was exposed to as a child:
"We Can Do All What You Can't Do."
I'm the bug you can't kill.

And I am not here for the *prestige*. Nor even for the *profit*.
I am here for our **community**.

In the Lulzboat, salute, bitch, and show some respect.[1]

YOU, you would have been that *lamer*, that kid who just wanted to show off and launched Exploit-DB PoCs from his bedroom like a *rookie*. To prove yourself, to put yourself on display, to make us see just how *in* you are. ME, I'm that kid who absorbed everything, and grew up with it.

Fucking *skids*.
Fucking normies!

Here's who we are: from Zone-H to respectable life.
From the underground sewers' toilets of the internet to a life where one manages to be quietly useful.
You would never have survived things like that.

From Hell to where we are today.
Saved by Research, by beautiful values, by thankless work. Not the kind that makes you shine. The kind passed on by those who know how to detect potential and save it before it's too late.

NOBODY saved us, we saved OURSELVES.
Thanks to what SCIENCE says.

We could have been bloodthirsty, instead we became skill-thirsty. We are people who have dedicated their lives to these screens, to this science, while you were bullying teenagers in Dota chats or doing god knows what, always showing off, always proving you're the most handsome, the strongest, the smartest, the most overbearing.

We, we are retarded, but we will stay online until the very end![2]

We are a true FAMILY!
And we are SLEEPLESS!

"It's completely autistic."
"You'll never amount to anything."
"You're crazy."
"What you're doing is completely stupid."
"Actually, what you write makes no sense whatsoever."

THE FUCK YOU THINK THIS IS?
You HOLLOW!
YOU HOLLOW, YOU UNDERSTAND ME?
YOU'RE WORTHLESS, YOU'RE FUCKING TRASH![3]

(btw: RIP, DG descendant...)

YOU'RE GOING TO BE REPLACED AND YOU'LL GO BACK TO HARASSING YOUR PEERS ON DOTA AND JERKING OFF TO VBA MACROS LIKE THE PIECE OF SHIT YOU ARE!
YOU FUCKING INCOMPETENT ASSHOLE!
YOU FUCKING PARASITE!
YOU FUCKING INCAPABLE!
YOU FUCKED WITH US!

Incorruptible

Moreover, my answer will be followed by a quote from David Heinemeier Hansson (DHH): "Fuck You."[4]

Yeah. Exactly: Fuck You. That's all.
I have NO INTEREST in programming that "way," and I OWN Ocarina.

Sharing Ocarina means **sharing a car I have maintained entirely myself, for myself, therefore with great care**. Still, it is shared as-is, and will remain so.
It's *my* car.

I channeled all my anger into it, to pour all my love into it.

Ocarina has a direction.
Those who wish to take it elsewhere with their *wishful thinking* are free to fork it and never contact me.

Contributing to an open source project because it's "cool" is a completely immature mindset, and tolerating this phenomenon causes real damage.

No genuine contributor does it for "fun" but out of *alignment of self-interests*.

Ocarina is not and will never be a *pontificating* solution.
Ocarina is and **WILL REMAIN a solution for solving real-world issues.**

If you disagree, **get back to r/unixporn[5], where you belong.** ✈️

First relevant interaction

A professional who had **actually** looked at the project structure reacted: "*This looks like one of my old Selenium projects and I hated that.*" That's **exactly** the kind of feedback Ocarina is built to address.

His first instinct was to equate POM with Selenium. Fair enough.
But POM works with any automation technology.
Is it "old school"? Absolutely. And? What's next?

Walking through his frustrations, he found out how Ocarina handles them: "Wait, that's it?"
But, this time, with a nod of respect. He admitted it was time to stop trying to be *the cleverest guy in the room*. Past a certain point, that's what kills a project.

The TRUE solidarity

Instead of going *nerdy*, Ocarina focuses on solving *small problems* without creating bigger ones.
The conclusion drawn from this is: "Ocarina is practical."
This is the purpose of Ocarina: its practicality.

This is how **we** work and that's really all it takes to get on board with Ocarina's philosophy.

No need to be a hacker, no need to have suffered so much.
It's a tool we offer, for the passionate ones, the truly passionate ones.

For those who want to build, not destroy.
For grounded, cultured people who know who they are.

For people like us, after all.
For once, let's unite.

Far from those who pillaged what we were.

Why not Playwright?

His follow-up: "Why not Playwright?"

Fair point.
Ocarina is agnostic, so wiring Playwright is easy.
The only constraint: Ocarina will never support `async`/`await`.

Secession

Today, people launch projects the way you'd pop out to buy a pack of cigarettes.
They try, because it's "trendy." So, they all copy each other, take out subscriptions to each other's products just to mutually inflate their numbers and deflate each other's *Stripe* dispute rate.

They think they're smart, but: the reality is that **every VC and every LP is well aware of it,** and each plays their part in a *fool's game*.

But there is something they will never understand. They have nothing "*underground*" about them: they are merely **kids in an identity crisis.** Children who were gently handed *Powerpoint* in exchange for their crayons.

A lesson for these amateurs: any truly "*cutting-edge*" project, whatever it may be, starts from a **pamphlet**, is rooted in **identity**, and goes through a stage of genuine **anti-marketing**. But since you're all *afraid of embarrassing your mother*, you never go through it.

As for me, I have nothing to lose in terms of reputation, there is no reputation to build under this name. Only a message to deliver, as it is, unfiltered.

It's time to kick all your *Juicero Presses* in the ass.
With that **knowing smile**, the one of those **who have finally managed to have what it takes to say STOP.**

IT is **what brought the smile back to our lives**, not what persecuted us through pseudo-"patterns" all more idiotic and inaccessible than the last.

You thought computer science was going to die?
Far from it.

Anti No-Code

Code is raw data. Auditable. Inspectable. **A white box.**
Exactly what AI has known how to work with since its very beginnings.

In our own silos, we'd already had a taste of this, even before *IntelliCode* (2018). As far back as 2013 on our end: a private *Emacs* (damn) plugin built by one of our mad scientists, R. Nobody could grasp his level in *reverse engineering*, nor his conciseness. Likewise in other areas... truth be told, he was bad at nothing, and had an extraordinary ability to always write *exactly what needed to be written*.

End of a myth: he had his own AI *autocomplete* and *auto-review*. 2013.

He would reject most of the generated code, but said this: "*I don't mind deleting 15,000 lines if it saves me from writing 1,000 myself.*"

By 2013, the game was no longer about being mindless "coding" apes.
It was about cultivating yourself to understand how a machine inspired by our own reasoning would help us rise above all that nauseating, doomed-to-die OOP.

But "savant" OOP won't be the first or the last casualty.
Computing evolves mostly in one direction: who would freely choose to use Windows 95 today?

And yet, λ-calculus (1930) is so powerful it's back in the spotlight today, just like AI (first formalization of the artificial neuron in 1943, McCulloch & Pitts).
Extraordinary in the history of IT.

The latest advances in AI have reshuffled the deck for SaaS, which exists only to hide complexity. Genuinely useful SaaS products, not the endless aggregators of this pompous "digital transformation," are always on the right side of History. However ugly they may be.

Companies from the *Prisma* or *Vercel* ecosystem spend ~$200 in tokens per *vendor* they drop. A weekend of *vibe coding*, and they wave goodbye to bloated, overpriced, sluggish SaaS platforms, in favor of *internal development* and a return to **raw data**.

December 2025: Lee Robinson, ex-Vercel, a familiar face from recent *Next.js* and *Turborepo* presentations among others, announces he's ditching *Sanity*, *Cursor*'s CMS choice until then. In favor of a return to **raw data**, to simple *Markdown* files. A few tokens, some well-executed *vibe coding*.

Thank you, goodbye.

Code is Law. Code *substitutes* for Law. Lessig, 1999, as a warning cry.
Then reclaimed by Ethereum in 2015, inversely, as a political ideal. A meaningful step forward, in a world of currency minted from factoring digits.

Anti devs

2016: the hack of The DAO, a decentralized investment fund on $ETH.
Why?

Bugs. Damned bugs, caused by damned devs.

An ideological regression of almost 20 years. The counter-power had made the lazy choice of "not understanding," the same choice that allowed a bunch of incompetents to slip under the radar.

And yet it's simple: unlike *Researchers*, engineers and their business school friends rely on *wishful thinking*.
The most "prestigious" schools taught them one thing: passing off hot air as "engineering," while those who actually know what they're doing call it *programming astrology*.

Anti startup nations

And once again, the grey-haired debate about "democratic governance of code."
But who is stepping in?

HR people?
CEOs?
Presales reps?

2022: ChatGPT. Yet three years later, startup founders are still selling *No-Code* "enhanced by AI."
Pure *cash burn* chasing a *market anomaly*, nothing more. Investors' *alpha*... in other words: their *casino*.

Don't name anyone, especially since their results are expected in Q2 2026.
Just say that on the basis of *fundamental analysis*, this bet makes no sense. That said, there's no need to wish them ill. It wouldn't be the first company we've seen fail miserably, nor the last we'd see succeed to everyone's surprise.

The fact remains that code is the most sovereign asset we have, and a value proposition centered on taking it away from us makes us hold our nose. The real problem is the devs we let bang on their keyboards without knowing what they're doing, like trained monkeys. Today *Claude Code* outperforms 99% of them, and they're howling. *The dogs bark, the caravan moves on.*

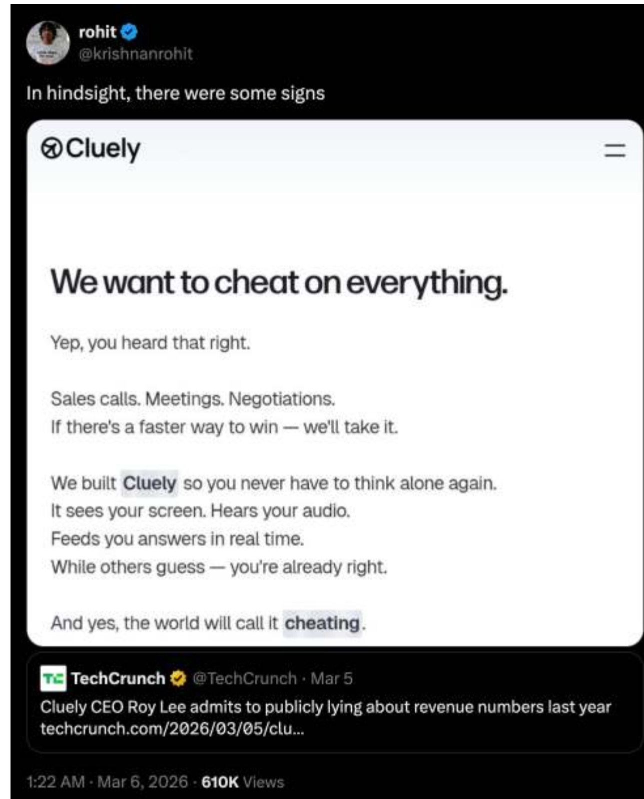
The true human value-add has never been more clearly identified: *taste* and *sense of responsibility*, qualities conspicuously absent in these "professionals" who drown us in entire "ideologies," just like those who *sell* what they don't understand.

Anti hype

That same December 2025, the startup whose name shall go unspoken was featured in certain media outlets as "*the AI No-Code test that humiliates Selenium and has BrowserStack shaking.*" No less.
Good luck to them: entrepreneurship is a world of bets and randomness, where the best ideas end up in the graveyard just as often as the most outlandish ones. Fundamental analysis gets beaten there regularly.

None of this, however, will steer Ocarina away from a vision that has been taking shape for over 15 years, far from the arrogance of all these *children of the hype*.

Whatever happens, we still wish them a fate that spares them from a humiliation as swift and spectacular as the one suffered by the founder of *Cluely* and we invite them to reconsider their arrogance.



Cluely CEO Roy Lee admits to publicly lying about revenue numbers last year. Then, Krishnan Rohit outlines that their value proposition was fraud since the beginning.

Today, the challenge of being the biggest scammer is becoming less and less profitable.
And we are glad of it. Once again, with AI, that sorely missing digital body, let us scream it, as loud as we screamed "**HACK THE PLANET**" back in '99: **CODE IS LAW.**

Anti slipologists

For too long, **software has been held hostage** by a minority, a "1%" who saw fit to turn it into a **playground for the initiated**: "geek tricks," "ninja techniques," "object-oriented." **The verdict is clear: it doesn't hold up, or barely does.**

What has been on our minds since the 1930s, **since the invention of λ-calculus**, is finally scalable to the degree we always wanted it to be.
Has our industry ever witnessed such an elegant formalization, one so deeply rooted in its own heritage?

Python's recent *type system* evolutions are central to what makes Ocarina possible.
Without being "the future" either: **this is established science, but one that arrived cruelly late in our ecosystem.**

As does AI, which is quietly rendering the "1%"'s capacity for harm obsolete.

The gratitude owed to these technological advances is as immense as the rage of those who rail against them, whom we fondly refer to in our parts as *slipologists*.

See also: *Haters*, by *Paul Graham*.[6]

For all these reasons, I have made the decision to **NEVER waste my time arguing when it's not worth it.** I enjoy arguing with people **focused on problem-solving and who have good judgment.**

For the rest: **make pointless PR red and pointless issues gray, no regret.**
Some people are simply not meant to please me, but so be it.

Ocarina is *the first of my projects that I'm opening to the public*, **but others are also on the way, including some far more interesting ones.**

From the underground,
For Now and Forever.

Peace out.[7]

> *"An idiot admires complexity, a genius admires simplicity, a physicist tries to make it simple. For an idiot anything the more complicated it is the more he will admire it, if you make something so clusterfucked he can't understand it he's gonna think you're a god cause you made it so complicated nobody can understand it. That's how they write journals in Academics: they try to make it so complicated people think you're a genius."*
>
> *— Terry Davis, "the smartest programmer that's ever lived"*

> *"To attain knowledge, add things every day. To attain wisdom, remove things every day."*
>
> *— Laozi*

---

[1] https://www.youtube.com/watch?v=PCW6BkSp1Sc
[2] https://soundcloud.com/spokepp4l/retarded
[3] https://soundcloud.com/queed-inc/true-colors
[4] https://world.hey.com/dhh/i-won-t-let-you-pay-me-for-my-open-source-d7cf4568
[5] https://www.reddit.com/r/unixporn/
[6] https://paulgraham.com/fh.html
[7] https://soundcloud.com/ytcracker/ytcracker-robots-will-definitely-take-your-job

Chapter 3

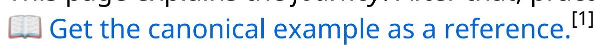
First steps

It's not about the destination, it's about the journey.

Disclaimer

> ***Note:*** *This book is intended to help you get familiar with the provided* `ocarina-example` *project, which remains* ***the source of truth*** *to refer to in all circumstances.*

> ⚠️ *The Ocarina Holy Book is NOT and will never be "plug-and-play." Ocarina requires a good level of maturity to use. As such, we will only focus on what can genuinely be tricky.*

This page explains the *journey*. After that, practice will be necessary.

📖 Get the canonical example as a reference.[1]

1. Project setup

Create a new Python project, then install the required dependencies:

```
pip install selenium
pip install ocarina
```

Then create your folders structure.

2. Adapters

Ocarina is built around a system of adapters that the user is responsible for writing. They allow the framework to be configured according to the constraints and conventions of each project.

The main adapters to create are:

- `act` *(required)*
- `test_campaign` *(required)*
- `test_suite` *(required)*
- `env_getters` *(optional)*
- `match_page` *(optional)*

2.1 EnvGetters

Ocarina's `EnvGetters` centralizes and types access to environment variables. It is divided into two categories:

- **Creds**: login/password pairs, expressed as immutable dictionaries.
- **Values**: individual values (strings).

```python
type _CredsKeys = Literal["dashboard"]
type _ValuesKeys = Literal["igor_xxx_key", "xxxxx_url"]


def _load_env() -> None:
    from dotenv import load_dotenv

    load_dotenv()


_DEFAULT_EFFECTS = (_load_env,)


class _EnvGetters(EnvGetters[_CredsKeys, _ValuesKeys]):
    def __init__(self, *, effects: Effects) -> None:
        for effect in effects:
            effect()

        super().__init__(
            credentials={
                "dashboard": MappingProxyType(
                    {
                        "login": os.environ["DASH_USERNAME"],
                        "password": os.environ["DASH_PASSWORD"],
                    }
                ),
            },
            values={
                "igor_xxx_key": os.environ["IGOR_XXX_KEY"],
                "xxxxx_url": os.environ["XXXXX_URL"],
            },
        )


def create_env_getters(*, effects: Effects | None = None) -> _EnvGetters:
    """Create a fresh EnvGetter instance."""
    if effects is None:
        effects = _DEFAULT_EFFECTS
    return _EnvGetters(effects=effects)
```

Once this adapter is in place, retrieving a value or credentials looks like this:

```python
xxxxx_url = create_env_getters().get_value("xxxxx_url")
dashboard_creds = create_env_getters().get_credentials("dashboard")
print(xxxxx_url)
print(dashboard_creds["login"])
print(dashboard_creds["password"])
```

> ***Note:*** *Valid keys are provided through two types:* `EnvGetters[_CredsKeys, _ValuesKeys]`*. If the user only wants to use* `.get_value()`*, it is enough to type* `_CredsKeys` *as* `Never`*. The same applies to* `_ValuesKeys`*, which should be typed as* `Never` *if the user only wants to use* `.get_credentials()`*.*

Our accessors are then strictly typed. For example:

```
xxxxx_url = create_env_getters().get_value("x")

# error: Argument 1 to "get_value" of "EnvGetters" has incompatible type
# "Literal['x']"; expected "Literal['igor_xxx_key', 'xxxxx_url']"
```

2.2 Act

In Ocarina, `act` is the verb used to express each single step in a test scenario. Its construction is intentionally left to the user, for reasons covered later in this book (*hooks*).

Its minimal shape is as follows:

```
def act(pom: TPOM, action: Callable[[TPOM], TPOM]) -> ActionStart[TPOM]:
    """Act on a page."""

    return create_act(
        pom,
        action,
    )
```

2.3 TestCampaign

The `TestCampaign` adapter is intentionally minimal. The only piece of information Ocarina cannot infer is the **number of workers**, i.e. the number of browsers to run in parallel for a given campaign. Since this parameter can also be passed directly via the CLI, a small adapter is all that is needed:

```
@final
class TestCampaign(OriginalTestCampaign[WebDriver]):
    """TestCampaign adapter."""

    def __init__(
        self,
        *,
        name: str,
        suites: Sequence[TestSuite[WebDriver]],
        max_workers: int | None = None,
        saturate_workers: bool | None = None,
    ) -> None:
        """Initialize the campaign."""
        if max_workers is None:
            max_workers = get_max_workers()

        super().__init__(
            name=name,
            suites=suites,
            max_workers=max_workers,
            saturate_workers=saturate_workers,
        )
```

> *The* `WebDriver` *type (Selenium or otherwise) is injected here:* `OriginalTestCampaign[WebDriver]`*.*
> *And here:* `suites: Sequence[TestSuite[WebDriver]]`

> ✅ *Of course, insert YOUR adapted* `TestSuite` *here, not Ocarina's built-in one.*

2.4 TestSuite

This is the most important adapter to understand. `TestSuite` natively exposes a large number of parameters. The goal of this adapter is to create a **facade** around it: some values are hardcoded once and for all, others are optionally exposed with sensible defaults. *Narrowing*.

Likewise:

```
@final
class TestSuite(OriginalTestSuite[WebDriver]):
    """TestSuite adapter."""

    def __init__(
        self,
        *,
        name: str,
        tests: Sequence[Test[WebDriver]],
        drivers_pool: SeleniumWebDriversPool,
        create_logger: Thunk[ILogger] | None = None,
        copy_indicator: str = "+",
        put_space_after_copy_indicator: bool = False,
        autoscreen_on_fail: bool = True,
        saturate_workers: bool | None = None,
    ) -> None:
        """Initialize the TestSuite."""
        if create_logger is None:

            def _create_logger():
                return create_matching_logger(get_logger_mode())

            create_logger = _create_logger

        super().__init__(
            name=name,
            tests=tests,
            only_ids=get_only(),
            exclude_ids=get_exclude(),
            max_retries_per_test=8,
            create_logger=create_logger,
            drivers_pool=drivers_pool,
            copy_indicator=copy_indicator,
            put_space_after_copy_indicator=put_space_after_copy_indicator,
            autoscreen_on_fail=autoscreen_on_fail,
            take_screenshot=_take_screenshot_on_fail,
            transient_errors=transient_errors,
            saturate_workers=saturate_workers,
        )
```

> *The* `WebDriver` *type (Selenium or otherwise) is injected here:* `OriginalTestSuite[WebDriver]`.
> *Also here:* `tests: Sequence[Test[WebDriver]]`
> *And here:* `drivers_pool: SeleniumWebDriversPool`

Transient errors

The concept of `transient_errors` is central to `TestSuite`.
These errors are treated as **noise**: if a test fails due to an exception listed in `transient_errors`, it is automatically replayed.
The maximum number of attempts is defined by `max_retries_per_test`.

This mechanism makes test execution tolerant to *flakiness*. Tests that replay frequently appear clearly in the logs, allowing maintainers to identify and fix sources of instability, whether caused by improper use of Selenium, out-of-scope environment conditions, or other external factors.

Only IDs and exclude IDs

These two parameters enable conditional test execution.
They are ID-based filters.

> ⚠️ ***Make sure to include them in this adapter, otherwise those CLI flags will not be handled.***

2.5 MatchPage

`match_page` is an Ocarina operator designed to handle pages with non-deterministic rendering: cookie banners, anti-bot challenges, A/B tests, etc.

Its logic is straightforward: **any exception raised is interpreted as a non-match, and therefore swallowed by** `match_page`. It is however possible to exclude some exceptions from this mechanism, so that they propagate normally up the execution flow.

For consistency, `transient_errors` should generally fall into this category: they must propagate rather than be silently swallowed.

The adapter is created as follows:

```
match_page = create_match_page(raised_exceptions=transient_errors)
```

3. Writing a first POM

The POM (*Page Object Model*) pattern being a well-established standard, we will not redefine it here.

Here is how to create a first POM with Ocarina:

```
@final
class Homepage(SeleniumTitleMixin, POMBase):
    """My homepage."""

    def __init__(
        self, *, driver: WebDriver, url: str = HOMEPAGE_URL
    ) -> None:
        """Initialize homepage POM."""
        self._driver = driver
        self._URL = url

    def open(self) -> Homepage:
        """Open the page."""
        self._driver.get(self._URL)
        return self

    def verify(self, *, timeout: float | None = None) -> Homepage:
```

```python
        """Verify function."""
        try:
            if timeout is None:
                timeout = get_timeout()

            WebDriverWait(self._driver, timeout).until(
                ec.title_is("Welcome to my homepage")
            )

            WebDriverWait(self._driver, timeout).until(
                ec.text_to_be_present_in_element(
                    (By.TAG_NAME, "h1"),
                    "My homepage",
                )
            )
        except TimeoutException as exc:
            raise PageVerificationError from exc

        return self
```

A few points are worth detailing.

3.1 SeleniumTitleMixin

Any object inheriting from `POMBase` must implement a `get_current_title` method. `SeleniumTitleMixin` provides this implementation transparently, without requiring it to be written manually.

Its role goes further: it also defines the `_driver` attribute with the `WebDriver` type (Selenium), making it **incompatible with any other type**. Attempting to assign an incorrect value will immediately produce a type error:

```python
self._driver = "lol"

# error: Incompatible types in assignment
# (expression has type "str", variable has type "WebDriver")
```

`SeleniumTitleMixin` therefore also acts as a **type guard**. Analogous mixins can be created for other browser automation technologies.

3.2 Returning `self`

Every action method returns `self`. This is a deliberate design choice in Ocarina, to be followed consistently: it enables method chaining and fluent scenario composition.

4. Writing connectors

Connectors are a thin but essential layer for scenario readability. They wrap POM method calls behind explicitly named functions:

```python
def open_homepage(p: Homepage) -> Homepage:
    """Open my homepage."""
    return p.open()
```

```python
def verify_homepage(p: Homepage) -> Homepage:
    """Verify we are on my homepage."""
    return p.verify()
```

They can also be composed directly:

```python
def open_then_verify_homepage(p: Homepage) -> Homepage:
    """Open my homepage, then verify it."""
    return p.open().verify()
```

5. Writing a first test scenario

All the building blocks are in place.
Here is how to assemble them into a scenario:

```python
def open_and_verify_homepage(driver: WebDriver, logger: ILogger):
    """Open and verify my homepage."""
    on_homepage = Homepage(driver=driver)

    just_log_error = create_just_log_error(logger=logger)
    just_log_success = create_just_log_success(logger=logger)
    log_error_with_current_url = create_log_error_with_current_url(
        logger=logger, driver=driver
    )
    log_success_with_current_url_and_take_screenshot = (
        create_log_success_with_current_url_and_take_screenshot(
            logger=logger, driver=driver
        )
    )

    return [
        drive_page(
            act(on_homepage, open_homepage)
            .failure(just_log_error("Failed to open the homepage..."))
            .success(just_log_success("Opened the homepage!")),
            act(on_homepage, verify_homepage)
            .failure(
                log_error_with_current_url(
                    "Failed to verify the homepage...",
                )
            )
            .success(
                log_success_with_current_url_and_take_screenshot(
                    "Verified the homepage!"
                )
            ),
        ),
    ]


test_homepage = create_selenium_test(
    name="Validate homepage",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=open_and_verify_homepage(driver, logger)
    ),
```

```
)
```

Each test step is expressed via `act`, to which a `.failure()` and a `.success()` handler are chained. The scenario is then wrapped in a `Test` object via `create_selenium_test`.

6. Creating a test suite

A suite groups a set of tests to be executed against the same driver pool:

```
def create_my_first_suite(
    *,
    drivers_pool: SeleniumWebDriversPool,
) -> TestSuite:
    """Create my first suite."""
    return TestSuite(
        name="My very first suite with Ocarina",
        tests=[
            test_homepage,
        ],
        drivers_pool=drivers_pool,
    )
```

7. Creating a test campaign

A campaign groups multiple suites:

```
def create_my_first_campaign(
    *, drivers_pool: SeleniumWebDriversPool
) -> TestCampaign:
    """Create my first campaign."""
    return TestCampaign(
        name="My very first campaign with Ocarina",
        suites=[
            create_my_first_suite(drivers_pool=drivers_pool),
        ],
    )
```

8. Creating a test cycle

A cycle groups multiple campaigns. It is the highest-level execution unit:

```
E2E_CYCLE_NAME = "My very first cycle with Ocarina"


def create_my_first_cycle(drivers_pool: SeleniumWebDriversPool):
    """Create my first cycle."""
    return TestCycle(
        name=E2E_CYCLE_NAME,
        campaigns=[
            create_my_first_campaign(drivers_pool=drivers_pool),
        ],
    )
```

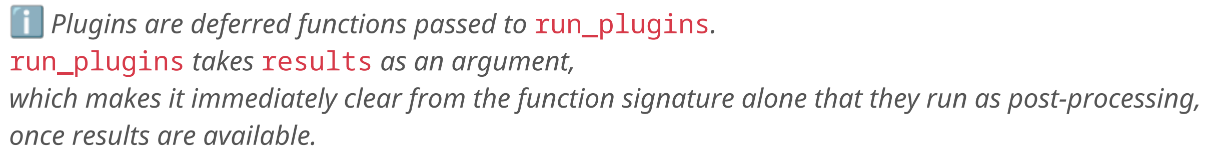
9. Bootstrapping the project

Here is the complete entry point for the project:

```
if __name__ == "__main__":
    CliStoreSingleton().push(create_selenium_auto_cli_store())

    drivers_pool = create_selenium_drivers_pool(
        browser=get_browser(),
        driver_path=get_driver_path(),
        headless=get_headless(),
        wait_timeout=get_timeout(),
        max_size=get_max_workers(),
        profile_path=get_profile_path(),
    )

    def _post_exec(results: TestCycleResults) -> None:
        print()
        pretty_print_results(results, with_colors=True)
        if has_test_cycle_failed(results):
            sys.exit(1)

    with timing(prefix="Tests duration:"):
        bootstrap(
            post_exec=_post_exec,
            test_cycle=create_my_first_cycle(drivers_pool),
            run_plugins=lambda results: run_plugins(
                lambda: generate_docx_proof(
                    logs_root=get_default_log_dir() / E2E_CYCLE_NAME,
                    logger=create_matching_logger(
                        "terminal"
                    ).set_domain_taxonomy(
                        ("Generate DOCX proofs plugin",)
                    ),
                    output_root=Path.cwd()
                    / ".reports"
                    / "tests_docx_output",
                ),
                lambda: generate_json_results(
                    results=results,
                    output_dir=Path.cwd()
                    / ".reports"
                    / "tests_json_output",
                    logger=create_matching_logger(
                        "terminal"
                    ).set_domain_taxonomy(
                        ("Generate JSON report file plugin",)
                    ),
                ),
                exceptions_logger=PrintLogger()
                .set_prefix(
                    lambda: concat_metadata(
                        format_utc_date_metadata_str,
                        format_current_thread_metadata_str,
                    )
                )
                .set_domain_taxonomy(("Post-execution plugins",)),
            ),
        )
```

The flow is as follows:

1. Arguments retrieved from the CLI are pushed into a global store.
2. A driver pool is created: it manages the lifecycle of web browsers running in parallel.
3. A `_post_exec` callback is defined: it runs after tests and plugins, prints the results, and exits with an error code if the cycle has failed.
4. Everything is bootstrapped inside a timer measuring the total execution duration. The execution flow is therefore: **cycle** → **plugins** → **post_exec**.



> ℹ️ *Plugins are deferred functions passed to* `run_plugins`.
> `run_plugins` *takes* `results` *as an argument,*
> *which makes it immediately clear from the function signature alone that they run as post-processing, once results are available.*

---

[1] https://github.com/mojo-molotov/ocarina-example

Chapter 4

First scenarios

Language is neither reactionary nor progressive; it is quite simply fascist; for fascism does not prevent speech, it compels speech.

The end of invalid states

... Make illegal states unrepresentable.

We will detail how `act`, `drive_page`, and `match_page` work when writing test scenarios with Ocarina.

Act and drive_page

Canonical example

Let's start with an example we will progressively break:

```
def go_from_homepage_to_book_call_page_with_the_cta(
    driver: WebDriver, logger: ILogger
):
    """Open and verify my homepage."""
    on_homepage = Homepage(driver=driver)
    on_book_a_call_page = BookCallPage(driver=driver)

    just_log_error = create_just_log_error(logger=logger)
    just_log_success = create_just_log_success(logger=logger)
    log_success_with_current_url_and_take_screenshot = (
        create_log_success_with_current_url_and_take_screenshot(
            logger=logger, driver=driver
        )
    )

    return [
        drive_page(
            act(on_homepage, open_then_verify_homepage)
            .failure(just_log_error("Failed to reach the homepage..."))
            .success(
                log_success_with_current_url_and_take_screenshot(
                    "On the homepage!"
                )
            ),
            act(on_homepage, click_book_call_page_cta)
            .failure(
                just_log_error(
                    "Failed to click on the 'Book a call' CTA..."
                )
            )
            .success(
                just_log_success("Clicked on the 'Book a call' CTA!")
            ),
        ),
        drive_page(
```

```
            act(on_book_a_call_page, verify_book_call_page)
            .failure(
                just_log_error(
                    "Failed to verify the 'Book a call' page..."
                )
            )
            .success(
                log_success_with_current_url_and_take_screenshot(
                    "On the 'Book a call' page!"
                )
            ),
        ),
    ]


test_homepage_book_a_call_cta = create_selenium_test(
    name="Go from homepage to book a call page, clicking the CTA",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=go_from_homepage_to_book_call_page_with_the_cta(
            driver, logger
        )
    ),
)
```

`drive_page` expresses that we are taking control of *one* page.
Every *transition* becomes explicit through the opening of a new `drive_page`.
Inside, `act` expresses an action emitted on that page: it is a *test step*. `drive_page` is variadic: it accepts as many `act` calls as needed, and the comma between each becomes an AND:

> *Open, then verify the homepage. AND click the CTA. We switch pages: verify that we are on the book-a-call page.*

Immune system

Let's try calling `verify_book_call_page` on `homepage`:

```
act(on_homepage, verify_book_call_page)

# error: Argument 2 to "act" has incompatible type
# "Callable[[BookCallPage], BookCallPage]";
# expected "Callable[[Homepage], Homepage]"
```

The action is incompatible with its target. This program does not *compile*.

Let's forget a `.success`:

```
drive_page(
    act(on_book_a_call_page, verify_book_call_page).failure(
        just_log_error("Failed to verify the 'Book a call' page...")
    )
)

# error: Expected type 'ActionSuccess[TPOM ≤: POMBase]',
# got 'ActionFailure[BookCallPage]' instead
```

Let's place a `.success` immediately after `act`:

```
drive_page(
    act(on_book_a_call_page, verify_book_call_page).success(
        log_success_with_current_url_and_take_screenshot(
            "On the 'Book a call' page!"
        )
    ),
),

# error:
# "ActionStart[BookCallPage]" has no attribute "success"
# Unresolved attribute reference 'success' for class 'ActionStart'
```

Let's swap `.success` and `.failure`:

```
drive_page(
    act(on_book_a_call_page, verify_book_call_page)
    .success(
        log_success_with_current_url_and_take_screenshot(
            "On the 'Book a call' page!"
        )
    )
    .failure(just_log_error("Failed to verify the 'Book a call' page...")),
),

# error:
# "ActionStart[BookCallPage]" has no attribute "success"
# Unresolved attribute reference 'success' for class 'ActionStart'
```

Let's chain heterogeneous `act` calls inside a single `drive_page`:

```
drive_page(
    act(on_homepage, open_then_verify_homepage)
    .failure(just_log_error("Failed to reach the homepage..."))
    .success(
        log_success_with_current_url_and_take_screenshot(
            "On the homepage!"
        )
    ),
    act(on_homepage, click_book_call_page_cta)
    .failure(just_log_error("Failed to click on the 'Book a call' CTA..."))
    .success(just_log_success("Clicked on the 'Book a call' CTA!")),
    act(on_book_a_call_page, verify_book_call_page)  # <- [!]
    .failure(just_log_error("Failed to verify the 'Book a call' page..."))
    .success(
        log_success_with_current_url_and_take_screenshot(
            "On the 'Book a call' page!"
        )
    ),
),

# error: Expected type 'ActionSuccess[Homepage]'
# (matched generic type 'ActionSuccess[TPOM ≤: POMBase]'),
# got 'ActionSuccess[BookCallPage]' instead
```

A lot of teams fight over *coding styles*.
Ocarina is more clear-cut: if the style is not followed, it is not a *lint* error, it is not a warning. It does

not **compile**.

Ocarina enforces the same template for everyone, and these errors surface directly in the editor via *mypy*: instant feedback.

The priority of test scenarios is their uniformity and simplicity. That's all.

match_page

`match_page` handles situations where a page can be rendered in different ways.

Let's start with the *matchers* principle:

```python
@final
class PageWithCookiesBannerMatchers:
    """Drive nuts anybody with this page or use matchers."""

    def __init__(self, *, driver: WebDriver) -> None:
        """Initialize helper."""
        self._driver = driver

    def has_cookies_banner(self) -> bool:
        """Quickly verify if the cookies banner is displayed."""
        timeout = min(get_timeout(), 5)

        try:
            WebDriverWait(self._driver, timeout).until(
                ec.visibility_of_element_located(
                    (By.CSS_SELECTOR, '[data-testid="cookies-banner"]')
                )
            )
        except TimeoutException:
            return False
        return True

    def has_not_cookies_banner(self) -> bool:
        """Quickly verify if the cookies banner is NOT displayed."""
        timeout = min(get_timeout(), 5)

        try:
            WebDriverWait(self._driver, timeout).until(
                ec.invisibility_of_element_located(
                    (By.CSS_SELECTOR, '[data-testid="cookies-banner"]')
                )
            )
        except TimeoutException:
            return False
        return True
```

A *matcher* minimally checks whether something is true, as fast as possible.

> ⚠️ *Still, avoid reaching for a raw* `.find_element(s)`*: it is the fast lane to Selenium flakiness.*

The 5s cap has no meaningful impact on a horizontally scaled test battery: it is not something to worry about here. It is also not recommended to disguise a `verify` as a *matcher*: **these are two different tools.**

Usage in a scenario:

```python
on_homepage = Homepage(driver=driver)
check_that_page = PageWithCookiesBannerMatchers(driver=driver)

# * ...
[
    match_page(
        branches=[
            when(
                check_that_page.has_cookies_banner,
                name="Has cookies banner",
                then=[
                    drive_page(
                        act(on_homepage, confirm_cookie_banner)
                        .failure(
                            log_error_with_current_url(
                                "Failed to click on the cookies banner's confirm button..."
                            )
                        )
                        .success(
                            log_success_with_current_url_and_take_screenshot(
                                "Clicked on the cookies banner's confirm button!"
                            )
                        )
                    )
                ],
            ),
            when(
                check_that_page.has_not_cookies_banner,
                name="Has NOT cookies banner",
                then=[],
            ),
        ],
        logger=create_matching_logger(
            "terminal"
        ),  # <- [!] If you want debug logs
    ),
    drive_page(act(on_homepage, ...).failure(...).success(...)),
]
```

`match_page` sits at the same level as `drive_page` and is chainable. Its `then` command expects a chain of `drive_page` or `match_page` calls. Branches are defined using `when`.

`match_page` and `when` were added late in Ocarina: the Igoristan was so unpredictable that the use case became obvious.
Their integration was straightforward, proof of the grammar's flexibility: other analogous structures could very well follow.

Repetitions

To repeat a test chain (e.g. to test multiple unauthorized access attempts), simply multiply the list:

```python
[
    drive_page(
        act(on_dashboard_welcome_page, click_on_go_to_nested_page_btn)
        .failure(
            just_log_error(
```

```
                "Failed to click on the go-to-nested-page button..."
            )
        )
        .success(
            just_log_success("Clicked on the go-to-nested-page button!")
        ),
        act(on_dashboard_welcome_page, verify_missing_otp_msg_is_displayed)
        .failure(
            just_log_error(
                "Failed to find the missing OTP auth message...",
            )
        )
        .success(
            log_success_with_current_url_and_take_screenshot(
                "Found the missing OTP auth message!"
            )
        ),
    ),
] * 5  # <- [!]
```

Fragments

A *fragment* is a `(driver, logger) -> TestChain` function that can be injected before or after the main chain, via `pre_test_scenarios_fragments` and `post_test_scenarios_fragments`.

For instance, `login_without_otp_happy_path` is a fragment:

```
def login_without_otp_happy_path(driver: WebDriver, logger: ILogger):
    """Verify that we can connect without OTP."""
    on_dashboard_login_page = DashboardLoginPage(driver=driver)
    on_dashboard_welcome_page = DashboardWelcomePage(driver=driver)

    # * ...
    return [
        drive_page(
            act(on_dashboard_login_page, open_dashboard_login_page)
            .failure(
                just_log_error(
                    "Failed to open the dashboard login page..."
                )
            )
            .success(just_log_success("Opened the dashboard login page!")),
            # * ...
        ),
        # * ...
    ]
```

Injecting at the beginning:

```
test_cant_access_the_protected_page_without_otp_using_the_ui = create_selenium_test(
    name="Can't access the protected page without OTP (using the UI)",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=dashboard_access_to_protected_page_without_otp_using_the_ui(
            driver, logger
        )
    ),
```

```
    pre_test_scenarios_fragments=[login_without_otp_happy_path],  # <- [!]
)
```

Injecting at the end:

```
test_dashboard_login_page_back_to_igoristan_button = create_selenium_test(
    name="Use the go back to Igoristan button",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=just_go_back_to_igoristan(driver, logger)
    ),
    post_test_scenarios_fragments=[verify_homepage],  # <- [!]
)
```

Both parameters can be combined and each accepts a list of *fragments*, injected in the order provided.

Aliasing

Scenarios can get heavy.
Since everything is declarative, the user is free to create aliases:

```
on_homepage = Homepage(driver=driver)
check_that_page = PageWithCookiesBannerMatchers(driver=driver)

click_confirm_cookies = drive_page(
    act(on_homepage, confirm_cookie_banner)
    .failure(
        log_error_with_current_url(
            "Failed to click on the cookies banner's confirm button..."
        )
    )
    .success(
        log_success_with_current_url_and_take_screenshot(
            "Clicked on the cookies banner's confirm button!"
        )
    )
)

# * ...
[
    match_page(
        branches=[
            when(
                check_that_page.has_cookies_banner,
                name="Has cookies banner",
                then=[click_confirm_cookies],  # <- [!]
            ),
            when(
                check_that_page.has_not_cookies_banner,
                name="Has NOT cookies banner",
                then=[],
            ),
        ],
        logger=create_matching_logger(
            "terminal"
```

```
        ),  # <- [!] If you want debug logs
    ),
    drive_page(act(on_homepage, ...).failure(...).success(...)),
]
```

Any value can be aliased and reused.
This writing is pure: it produces no immediate *effect*.
Everything can be redeclared elsewhere, reorganized elsewhere, as long as the final chain matches what is expected.

Chapter 5

First jutsus

And under ice, a river glitters...

Datasets

Driving a test with a dataset is straightforward with Ocarina:

```
multi_login_dataset: Sequence[
    MappingProxyType[ImmutableCredentialsKeys, str]
] = [
    MappingProxyType(
        {
            "login": "any",
            "password": "figatellu",
        }
    ),
    MappingProxyType(
        {
            "login": "Napoleon",
            "password": "figatellu",
        }
    ),
    MappingProxyType(
        {
            "login": "NoSicilianAllowed",
            "password": "figatellu",
        }
    ),
    MappingProxyType(
        {
            "login": "anonymous",
            "password": "figatellu",
        }
    ),
    MappingProxyType(
        {
            "login": "TheEmpire",
            "password": "figatellu",
        }
    ),
]


def _create_login_scenario(
    credentials: ImmutableCredentials,
) -> SeleniumTestScenario:
    """Welcome to functional factories."""

    def _scenario(driver: WebDriver, logger: ILogger):
        dashboard_creds = credentials  # <- [!] Provided by the closure

        on_dashboard_login_page = DashboardLoginPage(driver=driver)
```

```
        on_dashboard_welcome_page = DashboardWelcomePage(driver=driver)

        # * ...
        return Scenario(
            test_chain=[
                drive_page(
                    # * ...
                    act(
                        on_dashboard_login_page,
                        login_without_otp_and_with_retries(
                            dashboard_creds,  # <- [!]
                            retries_amount,
                            logger=logger,
                        ),
                    )
                    .failure(
                        just_log_error(
                            "Failed to connect to the dashboard without OTP...",
                        )
                    )
                    .success(
                        just_log_success(
                            f"Connected to the dashboard as {dashboard_creds['login']}!"
                            # ⬆️ [!]
                        )
                    ),
                ),
                drive_page(
                    act(on_dashboard_welcome_page, ...)
                    .failure(...)
                    .success(...)
                ),
            ]
        )

    return _scenario


multi_login_tests = [
    create_selenium_test(
        name=f"Login - {creds['login']}",
        test_scenario=_create_login_scenario(creds),
    )
    for creds in multi_login_dataset
]
```

A *closure*[1] is all it takes.
Note that `Scenario` is declared from the inside here. It makes sense, since the whole point is to encapsulate it.

`multi_login_tests` is therefore a *list* of `Test`, which we *unpack* into a `TestSuite`, like so:

```
def create_suite(
    *,
    drivers_pool: SeleniumWebDriversPool,
) -> TestSuite:
    return TestSuite(
        name="Login (data-driven PoC)",
```

```
        tests=[*multi_login_tests],
        drivers_pool=drivers_pool,
    )
```

Smoke tests

To run smoke tests at the start of a cycle with Ocarina:

```
E2E_CYCLE_NAME = "My very first cycle with Ocarina"


def create_e2e_test_cycle(drivers_pool: SeleniumWebDriversPool):
    """e2e test cycle."""
    return TestCycle(
        name=E2E_CYCLE_NAME,
        campaigns=[
            create_my_first_campaign(drivers_pool=drivers_pool),
        ],
        smoke_tests_campaigns=[
            create_my_first_smoke_campaign(drivers_pool=drivers_pool),
            create_my_second_smoke_campaign(drivers_pool=drivers_pool),
        ],
        mode="wait-for-all-smoke-tests",
    )
```

`mode` accepts two values (default: `"fail-fast-on-first-smoke-campaigns-sequence-fail"`):

- `"fail-fast-on-first-smoke-campaigns-sequence-fail"`: as soon as one smoke tests campaign fails, the remaining ones are skipped.
- `"wait-for-all-smoke-tests"`: all smoke tests campaigns run to completion, even if one fails along the way.

In both cases, main tests are skipped if any smoke test has failed.

Setup and teardown

`Scenario` accepts two optional callbacks: `setup` and `teardown`.

```
Scenario(
    setup=seed_test_user,
    test_chain=[
        drive_page(
            act(page, open_page)
            .failure(log_error("Failed to open..."))
            .success(log_success("Opened!")),
        ),
    ],
    teardown=delete_test_user,
)
```

Lifecycle

1. `setup()`: runs before the `test_chain`. On failure, the `test_chain` is skipped and `teardown` still runs. If every attempt fails due to setup, the test is marked **SKIPPED** (not **FAILED**),

2. `test_chain`: the actual test steps,
3. `teardown()`: always runs, even on failure. Errors are logged and ignored.

`setup` and `teardown` are `Effect`.
They are meant for infrastructure concerns: seeding a database, calling an API, cleaning up state...

If encapsulation is needed: *closure*.

Proxy pattern

One use case from the canonical example[2] is `HumanizedDriver`:

```
class HumanizedDriver(WebDriver):
    def __init__(
        self, driver: WebDriver, **keyboard_config: Unpack[KeyboardConfig]
    ) -> None:
        object.__init__(self)
        self._driver = driver
        self._config = keyboard_config

    def find_element(
        self,
        by: str | RelativeBy = "id",
        value: str | None = None,
    ) -> _HumanizedWebElement:
        element = self._driver.find_element(by, value)
        return _HumanizedWebElement(element, self._config)

    def find_elements(
        self,
        by: str | RelativeBy = "id",
        value: str | None = None,
    ) -> list[WebElement]:
        elements = self._driver.find_elements(by, value)
        return [_HumanizedWebElement(el, self._config) for el in elements]

    def __getattr__(self, name: str):
        return getattr(self._driver, name)
```

The idea: return *Web Elements* that behave differently for user-like interactions. Keystrokes, in this case.
Transparent to the *type system*, transparent to the *runtime*.

Which then allows:

```
create_selenium_test(
    name="Send the form",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=_send__form(
            HumanizedDriver(  # <- [!]
                driver,
                wpm=125,
                typo_rate=0.14,
                hesitation_rate=0.02,
                burst_rate=0.35,
                late_correction_rate=0.6,
            ),
```

```
            logger,
        ),
    ),
)
```

Or, with a *closure*:

```
def _scenario(driver: WebDriver, logger: ILogger):
    humanized_driver = (
        HumanizedDriver(  # <- [!]
            driver,
            wpm=125,
            typo_rate=0.14,
            hesitation_rate=0.02,
            burst_rate=0.35,
            late_correction_rate=0.6,
        ),
    )

    on_some_form_page = SomeFormPage(driver=humanized_driver)  # <- [!]
```

The same principle applies to the logger, routing it toward a *sink*, for instance. That case isn't canonically covered by Ocarina.

Reactive programming: NO

Ocarina test scenarios are intentionally static.
Yet a web application is dynamic, and sometimes capturing a value on the fly to pass it to a later step is perfectly legitimate.

Ocarina doesn't answer that. It doesn't need to.

Architectural answer

What we're after here is an *in-memory cache*.

We generate keys just before the test chain kicks off, and pass them to the POM actions. Actions write and read through a unique key.
The scenario just hands them out:

```
# * ...
cache = in_memory_cache_with_30m_ttl
username_key = reserve_free_cache_key(cache)
otp_send_date_key = reserve_free_cache_key(cache)

return [
    drive_page(
        # * ...
        act(
            on_dashboard_login_page,
            start_to_login_with_otp_and_with_retries(
                dashboard_creds,
                retries_amount,
                cache=cache,
                logger=logger,
```

```
                username_key=username_key,
                otp_send_date_key=otp_send_date_key,
            ),
        )
        .failure(
            just_log_error(
                "Failed to fill and confirm the login form with OTP...",
            )
        )
        .success(
            just_log_success(
                "Filled and confirmed the login form with OTP!"
            )
        ),
        act(
            on_dashboard_login_page,
            verify_otp_screen,
        )
        .failure(
            just_log_error(
                "Failed to verify the OTP screen...",
            )
        )
        .success(just_log_success("Verified the OTP screen!")),
        act(
            on_dashboard_login_page,
            type_otp_with_retries(
                retries_amount,
                cache=cache,
                logger=logger,
                username_key=username_key,
                otp_send_date_key=otp_send_date_key,
            ),
        )
        .failure(
            just_log_error(
                "Failed to confirm the OTP code...",
            )
        )
        .success(just_log_success("Confirmed the OTP code!")),
    ),
    # * ...
]
```

API calls and locks

API and locks have to be handled in the POMs.

>
> ⚠️ *Ocarina doesn't support `async`/`await` and never will.*

API calls: synchronous `requests` is enough.
Locks: `threading.Lock` if a single process at a time, otherwise Redis distributed locks are enough (`redis.StrictRedis` + `redis.lock`).

Browser profile

Some cases require passing a profile with `--profile-path`:

- **Proxy authentication**,
- **Pre-loaded extensions**,
- **Local settings** (language, timezone, certificates...),
- Etc.

---

[1] https://en.wikipedia.org/wiki/Closure_(computer_programming)
[2] https://github.com/mojo-molotov/ocarina-example

Chapter 6

First real-world hurdles

The only sensible thing was to adapt oneself to existing conditions.

Random server errors

Random server errors are a tough nut to crack.
During a particularly unpleasant work experience, I had to deal with an environment that regularly displayed totally random 500 error pages, regardless of which part of the application was being explored.

In cases like this, Ocarina's answer lives directly in the creation of the `act` verb:

```
ERROR_PAGE_REGEX = re.compile(r"^\d{3}(?!\d)")


@final
class HttpErrorPageReachedError(Exception):
    """Raised when error page is reached."""


def act(pom: TPOM, action: Callable[[TPOM], TPOM]) -> ActionStart[TPOM]:
    """Act on a page."""

    def failure_hook(pom: TPOM, exc: Exception) -> Fail:
        with suppress(Exception):
            title = pom.get_current_title()
            is_http_error_page = title and ERROR_PAGE_REGEX.match(
                title.strip()
            )
            if is_http_error_page:
                http_error = HttpErrorPageReachedError(
                    f"HTTP error page: {title}"
                )
                http_error.__cause__ = exc
                return Fail(error=http_error)
        return Fail(error=exc)

    return create_act(
        pom,
        action,
        on_failure=failure_hook,  # <- [!]
    )
```

The `on_failure` hook was designed precisely for this.
All it takes is creating some *guards* and modifying the error wrapped inside `Fail` to trigger a *retry* of any test that failed due to an external cause.

The next step should look familiar:

```
transient_errors = (
```

```
    HttpErrorPageReachedError,  # <- [!]
    PageVerificationError,
    # * ...
)

@final
class TestSuite(OriginalTestSuite[WebDriver]):
    """TestSuite adapter."""

    def __init__(
        self,
        *,
        # * ...
    ) -> None:
        """Initialize the TestSuite."""
        # * ...
        super().__init__(
            # * ...
            max_retries_per_test=8,
            transient_errors=transient_errors,
        )
```

Finally, if `match_page` is also used in the project and a shared `transient_errors` variable is undesirable, don't forget to add these new error definitions to the `raised_exceptions` parameter of the `match_page` constructor.

The automated test logs will provide a solid starting point for raising these issues with the team.

Random errors within a step

Thinking I had left that kind of nonsense behind me, I moved on, only to find unstable forms and authentication systems that worked half the time.

Here, Ocarina's answer is different: we delegate the responsibility to the POM.

```
@final
class CorsicamonEnterXXXKeyPage(SeleniumTitleMixin, POMBase):
    """Igoristan's corsicamon enter XXX key page."""

    def enter_xxx_key(self) -> CorsicamonEnterXXXKeyPage:
        """Enter XXX key."""
        # * ...

    # * ...
    def enter_xxx_key_with_retries(
        self, *, retries: int, logger: ILogger
    ) -> CorsicamonEnterXXXKeyPage:
        """Enter XXX key (n retries)."""
        validate(retries, name="retries").assert_that(
            is_positive
        ).execute().raise_if_invalid()

        attempts_count = 1
        self.enter_xxx_key()

        while attempts_count <= retries:
            timeout = get_timeout()
```

```python
            with suppress(Exception):
                WebDriverWait(self._driver, timeout).until(
                    ec.invisibility_of_element_located(
                        self._corsicamon_network_error_container
                    )
                )
                break

            msg = (
                "Failed to enter the XXX Key."
                "\n"
                f"Life: {attempts_count}/{retries}"
                "\n"
                f"Current URL: {self._driver.current_url}"
            )

            logger.warning(msg)
            take_screenshot(
                driver=self._driver, logger=logger, category="WARNING"
            )
            self.click_retry_button()
            attempts_count += 1

        s = "s" if attempts_count > 1 else ""
        msg = f"Entered the XXX Key. After {attempts_count} attempt{s}."

        logger.info(msg)
        return self
```

This also raises a question about *connectors*: how to pass parameters to them?

```python
"""Functional connectors."""

# * ...


def enter_xxx_key(
    p: CorsicamonEnterXXXKeyPage,
) -> CorsicamonEnterXXXKeyPage:
    """Enter the XXX key."""
    return p.enter_xxx_key()


# * ...


def enter_xxx_key_with_retries(
    *,
    retries: int,
    logger: ILogger,
) -> Callable[[CorsicamonEnterXXXKeyPage], CorsicamonEnterXXXKeyPage]:
    """Click on the retry button."""

    def unwrapped(
        p: CorsicamonEnterXXXKeyPage,
    ) -> CorsicamonEnterXXXKeyPage:
        return p.enter_xxx_key_with_retries(retries=retries, logger=logger)
```

```
    return unwrapped
```

Simply return the def with the expected signature, inside a function that captures the parameters. That's a *closure*[1].

Selenium random errors

Selenium offers plenty of opportunities to shoot yourself in the foot: *race conditions*, *stale element* errors, and so on.

The answer here is **pragmatic**: add `WebDriverException` directly to `transient_errors`, with a generous retry count (8, which means 9 lives, like a cat 🐱).

Capture all Selenium errors and watch the retries in the logs.
From there, it becomes easy to identify tests that could use some improvement.

Discrete random errors

Even more surprising: applications displaying error toasts for no apparent reason, or forms reporting validation errors on perfectly correct inputs, without actually blocking the flow.

These errors are the hardest to catch, precisely because they are *painless*. You can't simply notice a crash and add a *retry policy* while waiting for the bug to be fixed. They are, for all intents and purposes, invisible.

What's left? Butchering the test scenarios, or reaching for "ninja techniques."
Ocarina refuses both.

Use *watchers*:

```
def catch_me_if_you_can_cb(watcher: SeleniumWatcher) -> None:
    """Detect any element with CSS class 'catch-me-if-you-can' on the current page."""
    # NOTE: using JS here to bypass the implicit wait timeout.
    elements = watcher.driver.execute_script(
        "return Array.from(document.querySelectorAll('.catch-me-if-you-can'));"
    )

    if not elements:
        return

    raw = watcher.driver.execute_script(
        """
        return arguments[0].map(el => ({
            tag:       el.tagName.toLowerCase(),
            text:      el.innerText.trim(),
            id:        el.id,
            cls:       el.className,
            name:      el.getAttribute('name') || '',
            testid:    el.getAttribute('data-testid') || '',
        }));
        """,
        elements,
    )

    for attrs in raw:
        fingerprint = ":".join(
```

```
            filter(
                None,
                [
                    attrs["tag"],
                    attrs["text"],
                    attrs["id"],
                    attrs["cls"],
                    attrs["name"],
                    attrs["testid"],
                ],
            )
        )

        if fingerprint in watcher.cache:
            continue

        watcher.cache.add(fingerprint)
        watcher.report(
            f"catch-me-if-you-can element detected: <{attrs['tag']}> {attrs['text']!r}",
            label="CATCH_ME_IF_YOU_CAN",
        )


# * ...

test_send_chaotic_form = create_selenium_test(
    name="Send the chaotic form",
    test_scenario=lambda driver, logger: Scenario(
        test_chain=_send_chaotic_form(
            HumanizedDriver(
                driver,
                wpm=125,
                typo_rate=0.14,
                hesitation_rate=0.02,
                burst_rate=0.35,
                late_correction_rate=0.6,
            ),
            logger,
        ),
        watchers=[  # <- [!]
            create_selenium_watcher(
                callback=catch_me_if_you_can_cb,
                name="catch-me-if-you-can",
                poll_interval=0.8,
            ),
        ],
    ),
)
```

`catch_me_if_you_can_cb` is the *callback* that the *watcher* will invoke every 0.8 seconds (`poll_interval`).

Let's clarify a few things.

Using Javascript

The *watcher* is error-tolerant: it swallows exceptions silently.
There is therefore no benefit to using a Selenium function to grab a page element, it would only add unnecessary baggage.
Using Selenium's native functions would mean dealing with *implicit timeout* concerns.

Going straight through *Javascript* bypasses all internal *polling* logic and keeps the *watcher*'s execution as non-blocking as possible for the test running on the same *driver*.

The whole trick becomes invisible, as it only takes a few milliseconds.

Fingerprinting

Watchers expose a simple string cache, designed specifically for this need: if the same error stays visible and is detected every 0.8 seconds, there's no point seeing it show up repeatedly in screenshots, reports, and logs. The fingerprint lets you ignore what you've already seen.

Report

At the end of the *callback*, it calls: `watcher.report`.
This call manages:

1. Logging the friction detected by the *watcher*,
2. Taking a screenshot as a trace of what was detected.

HumanizedDriver

Nothing stops us from attaching behaviors to the *logger* or the *driver*. Here, since the form is temperamental, we opt for a slow, "humanized" test: typing with typos, corrections, hesitations. We simply wrap the *driver* in a *proxy*, `HumanizedDriver`.

Concurrency Heisenbugs

My quest was not over yet.
I once watched colleagues count to three before all clicking at the same time to trigger the same action, right there in the office. I found myself questioning the meaning of my life. And yet, by doing so, they genuinely managed to reproduce bugs.

This behavior can be reproduced with Ocarina.
By default, Ocarina is aggressive.

Its `saturate_workers` option forces random test cloning within a suite.
Whenever there are more *workers* available in the *DriversPool* than tests to run in a suite, Ocarina will randomly clone tests, spin up all drivers, and assign each of them a test to execute.

This option can be enabled from the `bootstrap` function. It can also be toggled individually, either at the suite level or at the campaign level.
When there is a conflict, the deepest element in the hierarchy has the final say.
For instance, if a campaign disables the option but a suite enables it, the suite takes priority.

```
if __name__ == "__main__":
    with timing(prefix="Tests duration:"):
        bootstrap(
            saturate_workers=False,  # <- True by default
```

```python
            # * ...
        )


# * ...
def create_campaign(
    *, drivers_pool: SeleniumWebDriversPool
) -> TestCampaign:
    return TestCampaign(
        saturate_workers=True,  # <- 'None' by default (cascade)
        max_workers=16,  # <- 'None' by default (CLI value)
        # ⬆️ Forcing saturate workers policy and 16 workers on this
        # campaign.
        # * ...
    )


# * ...
def create_suite(
    *,
    drivers_pool: SeleniumWebDriversPool,
) -> TestSuite:
    return TestSuite(
        saturate_workers=False,  # <- 'None' by default (cascade)
        # ⬆️ Will take the priority: saturate workers disabled on this
        # suite.
        # * ...
    )
```

It is also possible to temporarily create a suite with a single test to maximize targeting precision. Or to run the cycle across multiple machines simultaneously (horizontal scaling).

Beyond concurrency concerns, this cloning mechanism also aims to ensure that passing tests owe nothing to chance. This effect is amplified by the degree of horizontal scaling and the number of workers involved.

---

[1] https://en.wikipedia.org/wiki/Closure_(computer_programming)

Chapter 7

Extensibility

If we don't believe in freedom of expression for people we despise, we don't believe in it at all.

ValidationChain

Usable in POMs, `validate` allows expressing invariants as chains. Execution is **deferred**: `.execute()` must be called explicitly.

The result exposes `is_valid`, `errors`, and `validated_values`. It is inert by default. `.raise_if_invalid()` throws the exception if needed.

```
validate(checkbox.is_selected(), name="checkbox_is_selected").assert_that(
    is_truthy, msg="Couldn't select the OTP checkbox."
).execute().raise_if_invalid()
```

Chaining invariants

Multiple assertions on the same value:

```
validate(unsafe_min_date, name="cached_min_date")
.assert_that(is_str)
.assert_that(is_iso_utc_date_string).execute().raise_if_invalid()
```

Chaining validations

Multiple validations on different values:

```
chain_validations(
    validate(unsafe_username, name="cached_username").assert_that(is_str),
    validate(unsafe_min_date, name="cached_min_date")
    .assert_that(is_str)
    .assert_that(is_iso_utc_date_string),
).execute().raise_if_invalid()
```

Reusable invariants

To factor out a recurring validation, create an *Invariant Validator*:

```
def _workers_amount_chain(
    chain: ValidationStartBlock[int],
    value: int,
) -> ValidationAssertBlock[int]:
    msg = f"Value Error: Number of workers must be at least 1 (got: {value})."
    return chain.assert_that(is_positive, msg=msg).assert_that(
        is_not_zero, msg=msg
    )
```

```python
def validate_workers_amount(
    *, workers_amount: int, name: str
) -> ValidationAssertBlock[int]:
    """Validate that workers amount is at least 1."""
    return FrameworkInvariantValidator.create(
        workers_amount, name, _workers_amount_chain
    )


# * ...
validate_workers_amount(
    workers_amount=max_workers, name="max_workers"
).execute().raise_if_invalid()
```

Convention: `FrameworkInvariantValidator.create` for technical invariants, `BusinessInvariantValidator.create` for business invariants.

Custom assertions

Without argument:

```python
def is_str(value: Any) -> None:
    if not isinstance(value, str):
        msg = "Expected value to be string."
        raise InvariantViolationError(msg)
```

With argument:

```python
def is_equal_to(cmp: Any) -> Predicate[Any]:
    def unwrapped(value: Any) -> None:
        if value != cmp:
            msg = f"{value} is not equal to {cmp}."
            raise InvariantViolationError(msg)

    return unwrapped
```

Type safety

The type checker catches assertions incompatible with the value's type:

```python
validate("lol", name="n").assert_that(is_positive)

# error: Argument 1 to "assert_that" of "ValidationStartBlock" has
# incompatible type "Callable[[float], None]";
# expected "Callable[[str], None]"
```

Success and failure

`.success` and `.failure` each take an *effect* to execute.
The canonical example[1] implements several handlers: plain error logging, error logging with current URL, success logging, and success logging with screenshot (+ URL).

```python
def _append_current_url_in_msg(msg: str, driver: WebDriver) -> str:
    try:
        driver_healthcheck(driver)
        extended_msg = f"{msg}\nCurrent URL: {driver.current_url}"
    except DriverDiedError:
        extended_msg = (
            f"{msg}\nThe WebDriver is down, can't provide the current URL."
        )

    return extended_msg


def create_just_log_error(
    *, logger: ILogger
) -> Callable[[str], FailureHandler]:
    return lambda msg: lambda exc: logger.error(msg, exc=exc)


def create_log_error_with_current_url(
    *, logger: ILogger, driver: WebDriver
) -> Callable[[str], FailureHandler]:
    def unwrapped(msg: str) -> FailureHandler:
        def _log_error_with_url_effect(exc: Exception) -> None:
            extended_msg = _append_current_url_in_msg(msg, driver)
            return create_just_log_error(logger=logger)(extended_msg)(exc)

        return _log_error_with_url_effect

    return unwrapped


def create_just_log_success(
    *, logger: ILogger
) -> Callable[[str], SuccHandler]:
    def unwrapped(msg: str) -> SuccHandler:
        def _log_effect() -> None:
            logger.success(msg)

        return _log_effect

    return unwrapped


def create_log_success_and_take_screenshot(
    *, logger: ILogger, driver: WebDriver
) -> Callable[[str], SuccHandler]:
    def unwrapped(msg: str) -> SuccHandler:
        def _log_and_take_screenshot_effect() -> None:
            performed_dependent_effect = create_just_log_success(
                logger=logger
            )(msg)()
            take_screenshot(
                driver=driver, logger=logger, category="SUCCESS"
            )
            return performed_dependent_effect

        return _log_and_take_screenshot_effect

    return unwrapped
```

```python
def create_log_success_with_current_url_and_take_screenshot(
    *, logger: ILogger, driver: WebDriver
) -> Callable[[str], SuccHandler]:
    def unwrapped(msg: str) -> SuccHandler:
        def _log_success_with_url_and_take_screenshot_effect() -> None:
            return create_log_success_and_take_screenshot(
                logger=logger, driver=driver
            )(_append_current_url_in_msg(msg, driver))()

        return _log_success_with_url_and_take_screenshot_effect

    return unwrapped
```

Other handlers are worth considering:

- `create_log_error_with_retry_hint`: signals a *transient error* and therefore the possibility of flakiness,
- `create_log_error_and_send_alert`: sends a webhook on failure, without polluting the test itself,
- `create_log_success_and_record_timing`: captures an end timestamp to measure the actual duration of a step (to be combined with `on_run_effect` from `create_act`),
- Etc.

A *combinator* would also be worth considering.

Plugins

`bootstrap` allows post-execution plugins to run based on test cycle results. For instance, `generate_docx_proof` walks the log tree and generates one Word document (test proof) per test case, embedding screenshots and converting UTC timestamps to local time.

The idea: plugins reassemble artifacts produced along the way into a different form. A plugin generating a web dashboard report would be a natural fit, for example.

Extensible grammar

Test scenarios grammar is built on a single type: `ChainRunner[T]`. A scenario is a `list[ChainRunner]` executed sequentially, short-circuiting on the first failure. `drive_page` is just a thin wrapper around `chain_actions`, which builds a `ChainRunner`. Any function returning a `ChainRunner` plugs in without touching the framework.

`match_page` was added after the fact to handle variable-state pages (optional banners, A/B tests, maintenance pages…): it evaluates conditions in order and runs the first matching branch.

Another example would be `skip_if`: intentionally bypassing a portion of the scenario on a condition without failing (would return a neutral `Ok`), useful for environment or data-dependent optional steps.

The only extension contract: return a `ChainRunner`.

---

[1] https://github.com/mojo-molotov/ocarina-example

Chapter 8

Using Ocarina with AI

Hello it's me, your new best friend!

A working setup: a full test cycle built alongside Claude Code and Ocarina, against the public Katalon CURA demo.

📖 Get the AI example as a reference.[1]

The three spiritual stones

1. `CLAUDE.md` at the project root.
2. `skills/` with one `<name>/SKILL.md` per procedure.
3. Verification rule: every SUT claim comes from observation (probe, `gh api`, `curl -v`), never inference.

`CLAUDE.md`

Two variants. `CLAUDE.md` is full (rules + project layout, hierarchy, conventions, CI shape, PR template). `CLAUDE.slim.md` is rules only. Slim when context is heavy; full for onboarding and reviews. Full wins on disagreement.

Onboarding steps (venv, `pip install`, the skill battery copied into Claude Code, `ruff` / `mypy` / `pre-commit`, runner smoke-check) live in `setup-environment`.

Rules:

Security testing is functional and static, never active. No payloads, no crafted requests, no DevTools DOM manipulation. Black-hat scenarios go through the normal UI.

Use constants. Named values aren't inlined.

Datasets are human decisions. Proposing doesn't run.

Verify SUT behaviour empirically. Probe, `gh api`, or `curl -v`. Never inference. Re-derive each time: a probe answers only for what it ran; a prior diagnosis only for that run.

Each rule carries a one-line "*why*."

`skills/`

One Markdown file per skill, YAML frontmatter + body. Ten families.

Review (13)

Static reads; surface findings.

- `review-spec-gaps` — clarification questions on the FRD.
- `review-watcher-misuse` — `watcher.report(...)` against the negative-only convention.
- `review-compartmentalisation-leaks` — URLs, selectors, magic numbers out of place.

- `review-dead-code` — unused connectors / POMs / scenarios / suites / fragments / constants; per finding: delete, incubate (`<source-root>/incubator/`, dependency tree preserved), or keep.
- `review-report` — classify each FAIL / SKIP for one run.
- Plus: `review-type-ignore`, `review-match-candidates`, `review-unverified-transitions`, `review-submit-dispatchers`, `review-comment-drift`, `review-suite-stability`, `review-intent-collisions`, `review-watcher-emissions`.

Analyse (4)

- `analyse-flakiness` — widen the transient-error net; chronic deaths are real flakes.
- `analyse-fixture-flakiness` — instrument setup/teardown; surface cross-test contamination.
- `analyse-watcher-flakiness` — with/without each watcher, interval sweep.
- `analyse-screenshot-flakiness` — group by `(test, step, browser)`, spot differences.

Black-hat (6)

- `business-attack-ideation` — bring the product down.
- `incoherence-attack-ideation` — each step legal, the set impossible.
- `persistence-attack-ideation` — repeated retries on blocked actions.
- `permission-appropriateness-audit` — is the access model itself appropriate?
- `bfcache-exposure-ideation` — BFCache attacks.
- `lateral-resource-ideation` — IDOR via the address bar only.

Comprehend (4)

- `assess-test-base` — catalogue the suite.
- `assess-ecosystem` — bounded public research, token-budget capped.
- `understand-sut-constraints` — SUT bounds that break parallel tests.
- `understand-ocarina` — walk the docs.

Pick (3)

By mtime, never filename.

- `pick-screenshots`, `pick-logs`, `pick-reports`.

Author (8)

Each produces a deliverable.

- `empiricism` — verify before encoding; don't overwrite intentional-fail gap tests.
- `write-a-probe` — throwaway script, gitignored.
- `write-test-strategy` — generate the test-strategy doc from the suite (scope, types, coverage tables, cycle tree, pass/fail, gaps, CI matrix).

- `extend-coverage` — extend coverage from existing assets.
- `update-frd-and-tests` — propagate a spec update.
- `manual-reproduction-guide` — human-runnable repro.
- `manage-backlog` — `BACKLOG.md`.
- `pr-report` — PR-type-aware report.

Refactor (2)

- `refactor-fragmentation` — DRY per user preference.
- `introduce-pom-retries` — POM-internal retries with the two-test split (first-try + with-retries).

State (1)

- `question-state` — interrogate the environment before trusting a result.

Setup (1)

- `setup-environment` — venv, dev tooling, the Ocarina skill battery copied into Claude Code's skills directory, driver paths in `CLAUDE.local.md`, pre-commit loop, runner smoke-check.

Run (1)

- `propose-visual-review` — before a local dispatch, offer `--not-headless` (watch the browser play out) vs headless (CI-shaped). Composes the command; user runs.

Recurring chains

Suite isn't green: `review-report` → `analyse-*` → `write-a-probe` → finding lands in `IDENTIFIED_GAPS.md` / FRD / scenario comment → probe deleted.

Black-hat scenario looks promising: `empiricism` → `extend-coverage` (often intentional-fail).

Spec changes: `update-frd-and-tests` (FRD first, tests follow). Gap tests are reframed, not flipped.

New Ocarina primitive needed: `understand-ocarina` first, then writing.

About to dispatch a run: `propose-visual-review` — headed (`--not-headless`) or headless (CI-shaped)? Composes the command; user runs.

Discipline

Surface, don't apply. Skills produce; the user decides.

Empirical, not assertive. Every SUT claim is observed, cited, dated. Ritual phrase: *"Fair point, I'm assuming. Let me verify empirically."*

Gap tests are reframed, not turned green. Invert the assertion, rename, move the strategy-doc row, log the resolution in `IDENTIFIED_GAPS.md`. One motion via `update-frd-and-tests`.

Watcher emissions are negative signals only. A watcher emitting *"login succeeded"* breaks the contract.

Distributed when scarcity is shared. If workers contend on a SUT-capped resource (sessions, slots, quotas), coordinate through distributed primitives. Otherwise a worker-local in-memory cache is

fine — provided keys can't collide and generation is thread-safe.

Mtime, not filename. UUID suffixes are random; `pick-*` sorts by mtime.

What this setup isn't

- Doesn't generate tests autonomously.
- Doesn't patch hallucinations in CI; a failure triggers `review-report` + `analyse-*`.
- Doesn't rewrite the spec; only `update-frd-and-tests` does, with a revision line.
- Doesn't run active security tests. Ever.

Exposed resources

- https://mojo-molotov.github.io/ocarina-holy-book/llms.txt
- https://mojo-molotov.github.io/ocarina-holy-book/llms-full.txt
- https://mojo-molotov.github.io/ocarina-holy-book/CLAUDE.md
- https://mojo-molotov.github.io/ocarina-holy-book/CLAUDE.slim.md
- https://mojo-molotov.github.io/ocarina-holy-book/ocarina-ru.pdf
- https://mojo-molotov.github.io/ocarina-holy-book/ocarina-en.pdf
- https://mojo-molotov.github.io/ocarina-holy-book/ocarina-fr.pdf

[1] https://github.com/mojo-molotov/ocarina-with-ai-example