Le livre sacré d'Ocarina

Igor Casanova

Sommaire

Chapitre 1

Qu'est donc Ocarina ?

Les frameworks de test ont tous fait le même mauvais pari. Ocarina fait le contraire. Apprenez pourquoi.

Ocarina a été conçu pour permettre de créer **le plus facilement possible** des tests automatisés via navigateur web, en laissant un **contrôle total à son utilisateur**.

Encore un !

Eux

La plupart des frameworks de test ont été conçus dans un monde où la barrière entre "ceux qui codent" et "ceux qui définissent les tests" était réelle et structurelle.

Robot Framework, a essayé de la **contourner** avec un *DSL* (*Domain Specific Language*), incluant **leur propre format** et **leur propre écosystème de plugins**. Ainsi, RF impose de-facto ses propres standards : c'est le coût immédiat de sa promesse.

De la même sorte, *Cucumber* a essayé avec le *Gherkin* : un langage "naturel" qui, en pratique, **contraint tout le monde sans vraiment libérer personne**. Coût : **couche de traduction permanente, désynchronisation Gherkin/code**.

Tous ont parié sur la même chose : **masquer la complexité** pour réconcilier les profils.
Résultat : les non-techniques restent spectateurs, et les techniques se retrouvent **prisonniers d'un outil qu'ils n'auraient jamais choisi**.

Le coût le plus important est un **nivellement par le bas, la réduction des options et une "flexibilité" qui s'obtient en se *battant* contre des *outils* plutôt que d'utiliser des *solutions***.

Rien ne sert que chacun tire sur la corde qu'on lui tend si, au centre, il n'y a qu'un nœud gordien.

Nous

Mieux vaut être seul que mal accompagné.

Ocarina parie sur le contraire : cette barrière va disparaître. Il s'agit d'un non-débat sur lequel tout le monde s'est construit un nœud autour du cou. Outils honteusement compliqués, vendus comme "solutions". Impact : **désastre opérationnel** dès lors que l'on a un besoin qui ne peut pas s'exprimer dans un "cadre" qui n'est PAS réellement générique.

Et le pire : **toutes ces technologies continueront d'évoluer dans ce sens**. AUCUNE d'entre elle ne prendra ce *shift*, puisqu'il s'agit d'un changement de paradigme et d'**un retour aux fondamentaux qui contredit totalement leur proposition de valeur**.

Pourtant, ce qu'il nous reste à présent, c'est le besoin d'un code de test **lisible, traçable et flexible**, sous sa forme la plus **brute**.

Avec l'IA, et des outils comme *Claude Code*, ce pari devient chaque jour plus solide.
Le pont entre techniques et non-techniques n'est plus une couche d'abstraction.

C'est l'IA elle-même. IA qui travaille sur de la donnée brute.

Le problème culturel

Il y a un autre angle mort, rarement nommé : **la méthodologie**.

ISTQB, où es-tu ?

L'ISTQB et les testeurs professionnels ont construit, depuis des décennies, un vocabulaire précis et éprouvé : *cycles de test*, *campagnes*, *suites de test*, *cas de test*, *pas de test*. Une hiérarchie claire, pensée pour **organiser, tracer et piloter** la qualité logicielle.

Les outils automatisés, eux, ont **largement ignoré cet héritage**.

pytest, *Jest*, *Mocha*... tous sont **un mix hybride** où les testeurs doivent apprendre à penser comme des développeurs, et où personne ne parle vraiment la même langue.

Cette "méthode des deux langues" est un échec.

Retour aux fondamentaux

Ocarina ne fait pas ce compromis.

Sa structure est modélisée **directement et exclusivement** sur la méthodologie des testeurs. Chaque concept du code correspond à un concept métier du test. Pas d'emprunt, pas de détournement, pas de *"ça ressemble à peu près à"*.

Et parce qu'Ocarina prend ce pari jusqu'au bout : **il est entièrement autonome**. Pas de plugin pytest. Pas d'intégration forcée à un écosystème tiers. Ocarina est *batteries-included*, il n'a besoin de rien d'autre pour fonctionner et c'est un choix délibéré.

Le vrai point de douleur

De la "créativité"

Tout le monde s'acharne sur le *comment* : des interfaces, des abstractions, de "jolis" DSLs.
Pendant ce temps, le *pourquoi* disparaît.

Ocarina fait le choix inverse.

Toute sa conception ainsi que tout son livre sacré, sont focalisés sur le **pourquoi**, sur de réels problèmes.
Pas sur une abstraction à utiliser *"comme on aime"*.

Pour le *comment* ?
La réponse est simple : Ocarina est **dense, immédiatement opérationnel et strict**. Pensé pour que

les humains ainsi que les LLMs en comprennent le cœur et l'usage sans friction.

Que dit la science ?

Ici, tout repose sur des **fondations statiques** :

- typage,
- generics,
- programmation fonctionnelle.

Ocarina rend son mésusage difficile par conception : le compilateur fait foi.

Juste de l'algèbre.

> 📖 *Le mot "algèbre" vient de l'arabe الجبر (al-jabr), qui signifie "la réunion des parties brisées" ou "la réduction des fractures".*

La plupart des outils partent de la grammaire humaine pour soi-disant "formaliser".
Puis, on réalise qu'une machine ne peut pas la lire telle quelle.
Alors, on empile les "adaptateurs".

Nous n'appelons pas ça de la *formalisation* mais des *vues de l'esprit*.

Ocarina fait évidemment **l'inverse**.
C'est ce qui rend Ocarina **stable et extensible à la fois**.

Vision

Grammaire souveraine

Et si déléguer sa grammaire à des "standards" n'avait jamais été une bonne idée ?

Le vrai hiatus du test E2E **n'est pas d'imposer la "meilleure" novlangue**.
Réponse simple : Ocarina est **extensible**.

On y crée les verbes et conjonctions que l'on veut, le tout régi par des **règles strictes qui poussent l'ensemble à rester profondément cohérent**.

Il reste alors à garantir la **traçabilité et la robustesse**.

Né sur le terrain

Dans Ocarina, chaque étape est observable.
Le chemin d'erreur est explicite. Le rapport de test **émerge naturellement du code**.

Pas de surarchitecture. Pas de dépendances inutiles.
Juste quelque chose de **petit, lisible, et qui tient dans le temps**.

Le plus fort, c'est qu'**Ocarina n'invente rien : Ocarina en revient aux fondamentaux**.

Adoption

Pour les scénarios les plus extrêmes : Ocarina n'a pas besoin d'être installé.
Il se copie, s'adapte, et tourne.
Pas de dépendance à auditer.

Pour les équipes bloquées par des *politiques de sécurité* : le code est petit, auditable en une après-midi. Rien de caché. Les seules dépendances externes sont dans les plugins post-exécution et si l'une d'elles ne passe pas, elle se retire sans que le reste ne casse.

En pratique, un consultant peut arriver chez un client avec Ocarina dans sa poche, *presque* sans demander la permission à personne.

Ce ne sont pas les hommes qui fatiguent, mais les vieilles ficelles qui les usent.

Et c'est aussi pour ces raisons qu'il existe : pour rendre aux testeurs leur **indépendance**.
Le tout avec un **bijou de synthèse**.

Chapitre 2

Premiers retours

La première des critiques ayant été faite à Ocarina. Devinez laquelle.

Dès ses débuts, Ocarina a été présenté discrètement à des professionnels du milieu, de tous horizons.

Premiers griefs

La première "critique" a souvent été la même : *"Tu te vantes de quelque chose qui est très simple."*

Certains s'en sont arrêtés là.
D'autres ont voulu aller plus loin : essayer d'expliquer ce qu'était, selon eux, la "vraie complexité".

Mais **aucun d'entre eux n'aurait été capable de réaliser la même chose.**

Le plus amusant, **c'est que je n'ai pas eu besoin de parler de complexité moi-même**, mais **juste de présenter quelques scénarios de test écrits avec Ocarina** pour que **cette réaction émerge immédiatement.**

Ce n'est que l'expression du conditionnement opérant de 99% des personnes de notre industrie.

Encore ajouter un rouage de plus à une machine incompréhensible semble être le fétichisme de beaucoup d'ingénieurs Mojo.

Pour ceux-là, il a suffi d'ouvrir le capot.

Premièrement : il ne s'agit pas de se vanter.
Ocarina **n'est pas fruit de vantardise**, contrairement à ce que beaucoup "d'ingénieurs" produisent. Il est le simple fruit de ce qui *émerge naturellement*.

Le **narcissisme** s'exprime à travers le fait de tout rendre honteusement compliqué dans l'unique but de *dominer*. Pas à travers l'acquisition de compétences et la simplification des process, puisque la finalité est le dysfonctionnement total : le chaos.

C'est aussi à ça que l'on reconnaît les narcissiques rapidement dans un *casino* : plus une machine brille, plus elle semble "compliquée", plus un narcissique va vouloir s'y asseoir et montrer qu'il est "plus intelligent que la machine".

On les reconnaît aussi face à leur *intolérance à l'échec*. Ils veulent systématiquement montrer à quel point, EUX, ont "pensé à tout".

Jamais de rouge avec un narcissique, que du vert, que du "prêt à partir en production".

Les conséquences sont dramatiques en plus d'être radicalement opposées à ce qui est l'état d'esprit du test logiciel : le narcissisme n'apporte rien de bon dans ce domaine.

De toute manière, il n'apporte strictement rien de bon nulle part : il n'apporte rien si ce n'est *l'enfer sur terre*.

Ocarina est radicalement opposé à ces phénomènes.

Complexité ostentatoire

La vraie complexité ne se montre pas, elle se ressent.
Dans la stabilité, dans l'extensibilité, dans ce qui ne casse pas.

Il n'y a **aucune valeur ajoutée à créer quelque chose d'horriblement compliqué à utiliser**, si ce n'est démontrer que c'est compliqué. **Personne ne demande ça.**

Il y a un sérieux problème dans notre industrie avec la considération que l'on a pour la complexité comme gage de sérieux.

Cette complexité est **là pour "impressionner ses pairs"**, et n'apporte rien si ce n'est **un dépassement de soi dans le manque d'intelligence.**
Se surpasser dans la mauvaise direction est pire encore que la médiocrité ordinaire et là est bien le seul "exploit" à y voir : félicitations, vous êtes bien arrivés au sommet de la **pyramide de la connerie** où il n'existe plus aucune remise en question !

C'est aussi pour cette raison que **KISS** (*Keep it simple, stupid*) est si mal compris dans l'industrie : **beaucoup imaginent que "simple" signifie "peu élaboré"**.
Difficile de moins bien comprendre une théorie.
On se retrouve donc avec le pire des deux mondes.

Comment peut-on prétendre pouvoir être *productif* de cette manière ? Sérieusement ?

Vous. Ne. Savez. Même pas. Ce que. "Clean code". Veut. Réellement. DIRE.
Vous n'avez même pas essayé d'en faire !

Philosophie

KISS

KISS est au cœur d'Ocarina.

Et si, en lisant un premier scénario de test, la réaction est *"c'est super simple"*, il est dommage de penser le faire sur le ton de la critique : **c'est exactement le compliment recherché**.
Merci.

Pour en arriver là, la question n'a jamais été de "briller" plus que les autres.
Il ne s'agit d'ailleurs pas d'un réel challenge.
La question a juste été : **de quoi ai-je réellement besoin ?**

Certains confrères ont eu d'autres idées, quant à elles assez "créatives" :

- Tordre l'implémentation de ROP (*Railway Oriented Programming*) d'Ocarina jusqu'à lui en faire perdre tout son sens, puisqu'ils ne savaient même pas ce que signifie ROP,
- Inclure des "*hooks*" et autres "*techniques de ninja*" en plein milieu des pas de test,
- Abandonner le typage,
- Tout réécrire en Rust pour la "performance",
- M'imposer leur incompréhension de l'évaluation paresseuse et de l'*IoC* comme des vérités absolues,
- "M'expliquer" ce que sont la programmation impérative et déclarative en racontant n'importe quoi,
- Me "parler" d'*event-driven programming*, toujours en racontant n'importe quoi,
- Et pire que tout, me parler d'une horrifiante théorie d'"orienté objet déclaratif",
- Etc.

Même un envers qui j'avais beaucoup de respect jusqu'alors a commencé à me casser les couilles.
Il pensait avoir des choses à m'apprendre, commençait à être hautain, à me dire "ce qu'il manquait" d'un ton désagréable, alors que l'intégralité de ce dont il parlait était non seulement sur la *roadmap* bien qu'encore bien masqué stratégiquement, mais en plus en allant plus loin que tout ce qu'il pourrait s'imaginer.

Moi, je ne disais rien, je restais silencieux après avoir envoyé le *repo*.
Lui, immédiatement, il "savait tout" mieux que tout le monde.

C'est quoi être "disruptif"

> *Bah là je fais le YC et mon client c'est IBM.*

> *Oui et moi je connais la reine d'Angleterre.*

> *Je vous ai déjà parlé des yakuzas ?*

> *J'ai fait une grande école et depuis j'apprends plus rien.*

D'accord : moi je vais te citer la devise du premier groupe d'hackers indonésiens auquel j'avais été exposé dès mon enfance : "We Can Do All What You Can't Do".
Je suis le bug dont tu ne pourras jamais te débarrasser.

Et moi je ne suis pas là pour le *prestige*. Ni même pour le *profit*.
Je suis là pour notre **communauté**.

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

TOI, tu aurais été ce *lamer*, ce gamin qui aurait juste voulu frimer et lancer des PoC trouvés sur Exploit-DB depuis ta chambre comme un *rookie*. Pour te montrer, pour t'exhiber, pour nous faire voir à quel point tu es *in*. MOI, je suis ce gamin qui a tout absorbé, et qui a grandi avec ça.

Putains de *skids*.
Putains de *normies* !

Voilà qui nous sommes : de Zone-H à une vie rangée.
Des chiottes souterrains d'internet à une vie où l'on arrive à se rendre utile discrètement.
Tu n'aurais jamais survécu à des choses pareilles.

De l'Enfer à là où nous en sommes aujourd'hui.
Sauvés par la Recherche, par de belles valeurs, par le travail ingrat. Pas celui qui fait briller. Celui transmis par ceux qui savent détecter un potentiel et le sauver avant qu'il ne soit trop tard.

PERSONNE ne nous a sauvés, on s'est sauvé SEUL.
Grâce à ce que dit la SCIENCE.

Nous aurions pu être assoiffés de sang, à la place nous sommes devenus assoiffés de compétences. Nous sommes des gens qui ont dédié leur vie à ces écrans, à cette science, pendant que tu harcelais des ados dans des chats sur Dota ou que tu foutais on ne sait trop quoi, à toujours t'exhiber, toujours montrer que tu es le plus beau, le plus fort, le plus intelligent, le plus dominant.

Nous, nous sommes des attardés, mais nous resterons en ligne jusqu'au bout ![2]

Nous sommes une vraie FAMILLE !
Et nous ne dormons JAMAIS !

"C'est complètement autistique."
"Tu n'en feras jamais rien."
"Tu es fou."
"C'est complètement con ce que tu fais."
"En fait ce que tu écris ça n'a aucun sens."

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...)

ON VA TE REMPLACER ET TU RETOURNERAS HARCELER TES PAIRS SUR DOTA ET TE BRANLER SUR DES MACROS VBA COMME UNE MERDE !
PUTAIN D'INCOMPÉTENT QUI NOUS EMMERDE !
PUTAIN DE PARASITE !
PUTAIN D'INCAPABLE !
YOU FUCKED WITH US!

Incorruptibles

Ma réponse sera complétée d'une citation de David Heinemeier Hansson (DHH) : "Fuck You".[4]

Oui. Vraiment : Fuck You. Pas compliqué.
Je ne VEUX PAS programmer de cette "manière" et je suis le PROPRIÉTAIRE d'Ocarina.

Transmettre Ocarina, c'est **transmettre une voiture dont j'ai fait tout l'entretien moi-même, pour moi-même, donc très précautionneusement**. En revanche, elle est et restera transmise telle quelle. C'est *ma* voiture.

J'y ai canalisé toute ma colère, pour y mettre tout mon amour.

Ocarina a une direction.
Ceux qui souhaitent l'emmener ailleurs avec leurs *vues de l'esprit* sont libres d'en faire un fork et de ne jamais me contacter.

Contribuer à un projet open source parce que c'est "stylé" est une vision totalement immature et la tolérance à ce phénomène crée des ravages.
Aucun réel contributeur ne le fait pour le "fun" mais par *harmonisation des égoïsmes*.

Ocarina n'est pas et ne sera JAMAIS une solution *pontifiante*.
Ocarina est et **RESTERA une solution pour résoudre des problèmes concrets.**

En cas de désaccord, **un retour sur r/unixporn[5] s'impose, chacun chez soi.** ✈️

Première interaction pertinente

Un professionnel ayant **vraiment** examiné la structure du projet a réagi : "*Ça ressemble à un de mes anciens projets Selenium, et je n'avais pas aimé ça*". C'est ce retour qui tombe **pile** dans ce qu'Ocarina cherche à résoudre.

Son réflexe a été d'associer POM à Selenium. Compréhensible.
Mais POM est applicable avec n'importe quelle technologie.
Est-ce que c'est "*old school*" ? Absolument. Et alors ?

En échangeant sur ses frustrations, il a découvert comment Ocarina y répond : "Ah ? C'est tout ?", mais cette fois avec respect. Lui, a reconnu qu'il était bon d'arrêter de vouloir être *le plus malin*. Car passé un certain stade, c'est ce qui tue un projet.

La VRAIE entraide

Plutôt que de vouloir "faire des trucs de geeks", Ocarina propose de résoudre de *petits problèmes* sans en créer de plus importants.
La conclusion qui en a été tirée est : "Ocarina est pratique".
C'est le but d'Ocarina : sa praticité.

C'est dans cette ambiance que l'on travaille **ensemble**.
Il ne suffit que de ça pour commencer à partager la philosophie autour d'Ocarina.

Pas besoin d'être un hacker, pas besoin d'avoir autant souffert.
C'est un outil que l'on donne, pour les passionnés, les vrais passionnés.

Pour ceux qui ont envie de construire et non pas de détruire.
Pour les personnes saines, cultivées, qui savent qui ils sont.

Pour des gens comme nous, dans le fond.
Pour une fois, serrons-nous les coudes.

Loin de ceux qui ont pillé ce que nous étions.

Pourquoi pas Playwright ?

Son second retour : "Pourquoi pas Playwright ?".

Tout aussi pertinent.
Ocarina est agnostique, donc c'est un simple branchement différent à faire.
Une seule contrainte : Ocarina ne supporte pas `async`/`await` et ne le fera jamais.

Sécession

Aujourd'hui les gens lancent des projets comme on descend de chez soi acheter un paquet de clopes.
Ils essaient, parce que c'est "branché". Alors, ils se copient tous entre eux, se prennent tous des abonnements entre eux pour gonfler mutuellement leurs chiffres et dégonfler leurs taux de litige *Stripe*.

Ils se croient malins : la réalité est que **tout VC, tout LP, tout le monde, en est bien conscient** et chacun tient son rôle dans un *jeu de dupes*.

Mais il y a quelque chose qu'ils ne comprendront jamais. C'est qu'ils n'ont rien "d'underground" et ne sont que **des gamins en crise d'identité.** Des enfants à qui on a gentiment troqué leurs crayons de couleur contre *Powerpoint*.

Leçon à ces amateurs : n'importe quel projet "*cutting-edge*", quel qu'il soit, démarre d'un **pamphlet**, est **identitaire** et passe par une étape de réel **anti-marketing**. Mais comme vous avez *peur de faire honte à votre maman*, vous n'y passez jamais.

Quant à moi, je n'ai rien à perdre en termes de réputation puisqu'il n'y a pas de réputation à construire sous ce nom. Uniquement un message à livrer, tel qu'il est, sans filtre.

Il est l'heure de botter le cul de toutes vos *presses Juicero*.
Avec ce **sourire en coin**, celui de ceux **qui sont enfin parvenus à avoir de quoi dire STOP.**

L'informatique est ce qui nous a **redonné le sourire dans nos vies**, pas ce qui nous a persécuté à travers des soi-disant "patterns" tous plus idiots et inaccessibles les uns que les autres.

Vous pensiez que l'informatique allait mourir ?
Et bien non.

Anti No-Code

Le code est une donnée brute. Auditable. Consultable. **Une boîte blanche.**
Très exactement ce avec quoi l'IA sait travailler depuis ses débuts.

Dans nos silos, on y avait déjà goûté, avant même *IntelliCode* (2018). Dès 2013 de notre côté : un plugin *Emacs* (ouch) privé créé par l'un de nos savants fous, R. Personne ne comprenait son niveau en *reverse engineering*, ni sa concision. De même dans les autres domaines, à vrai dire, il n'était mauvais nulle part et avait une capacité extraordinaire à toujours écrire *exactement ce qu'il fallait écrire*.

Fin d'un mythe : il avait sa propre *autocomplétion* et *auto-review* IA. En 2013.

Il rejetait la majorité du code généré, mais disait ceci : "*Ça ne me dérange pas de supprimer 15 000 lignes si ça m'évite d'en écrire 1000 moi-même.*"

Dès 2013, l'enjeu n'était plus d'être des imbéciles de "codeurs".
Mais de se cultiver pour savoir comment une machine inspirée de notre raisonnement nous aiderait à nous démarquer de toute cette POO nauséabonde et vouée à mourir.

Mais la POO "savante" ne sera ni la première ni la dernière morte au combat.
L'informatique évolue majoritairement dans un seul sens : qui choisirait librement d'utiliser

Windows 95 aujourd'hui ?

Pourtant, λ-calcul (1930) est si puissant qu'il revient sur le devant de la scène aujourd'hui, tout comme l'IA (première formalisation du neurone artificiel en 1943, McCulloch & Pitts).
Extraordinaire dans l'Histoire de l'IT.

Les dernières avancées en IA ont rebattu les cartes du SaaS qui ne sert qu'à masquer la complexité. Les SaaS profondément utiles, pas les énièmes agrégateurs de cette "transformation numérique" pompeuse, sont toujours du bon côté de l'Histoire. Aussi moches soient-ils.

Des entreprises comme celles de l'écosystème *Prisma* ou *Vercel* investissent ~200$ de tokens par *vendor* dont elles se séparent. Un week-end de *vibe coding*, et elles disent adieu aux SaaS aux mille fonctionnalités, trop chers, trop inertes, au profit du *développement interne* et du retour à la **donnée brute**.

Décembre 2025 : Lee Robinson, ex-Vercel, figure des dernières présentations *Next.js* et *Turborepo* entre autres, annonce qu'il se débarrasse de *Sanity*, le CMS jusqu'alors utilisé chez *Cursor*. Au profit d'un retour à de la **donnée brute**, à de simples fichiers *Markdown*. Quelques tokens, du *vibe coding* bien exécuté. Merci, au revoir.

Code is Law. Le code se *substitue* à la Loi. Lessig, 1999, un cri d'alerte.
Puis repris par Ethereum, 2015, à l'inverse comme idéal politique. Une belle avancée, portée par ces monnaies forgées de la factorisation des nombres.

Anti dévs

2016 : hack de The DAO, fonds d'investissement décentralisé sur $ETH.
Pourquoi ?

Des bugs. Des satanés bugs, à cause de satanés dévs.

Retour idéologique de quasi 20 ans en arrière. Le contre-pouvoir avait fait le choix paresseux de "ne pas comprendre", même choix qui a permis à une bande de nuls de passer inaperçue.

C'est pourtant simple : contrairement aux *Chercheurs*, les ingénieurs et leurs amis d'école de commerce s'appuient sur des *vues de l'esprit*.
Les écoles les plus "prestigieuses" leur ont appris une chose : faire passer le melon pour de "l'ingénierie", là où ceux qui savent qualifient ce travail d'*astrologie de la programmation*.

Anti startup nations

Et à nouveau ce débat de cheveux gris sur la "gouvernance démocratique du code".
Mais qui y participe ?

Des RH ?
Des PDG ?
Des avant-ventes ?

2022 : ChatGPT. Mais, trois ans plus tard, des startupers persistent à vendre du *No-Code* "augmenté par l'IA". Du *cash burn* dans un espoir d'*anomalie de marché*, rien d'autre. L'*alpha* des investisseurs, autrement dit leur *casino*.

On ne citera personne, d'autant que leurs résultats sont attendus pour T2 2026.
Disons juste que sur la base de l'*analyse fondamentale*, ce pari n'a aucun sens. Il n'est pour autant pas nécessaire de leur interdire de vivre. Ce ne serait ni la première boîte que l'on verrait lamentablement échouer, ni la dernière que l'on verrait réussir avec stupéfaction.

Toujours est-il que le code est ce que l'on a de plus souverain, et qu'une promesse de valeur consistant à nous le retirer nous fait nous pincer le nez.
Le vrai problème, ce sont les dévs qu'on laisse taper sur leurs claviers sans savoir ce qu'ils font, comme des singes savants. Aujourd'hui *Claude Code* exécute mieux que 99% d'entre eux, et ils crient. *Les chiens aboient, la caravane passe.*

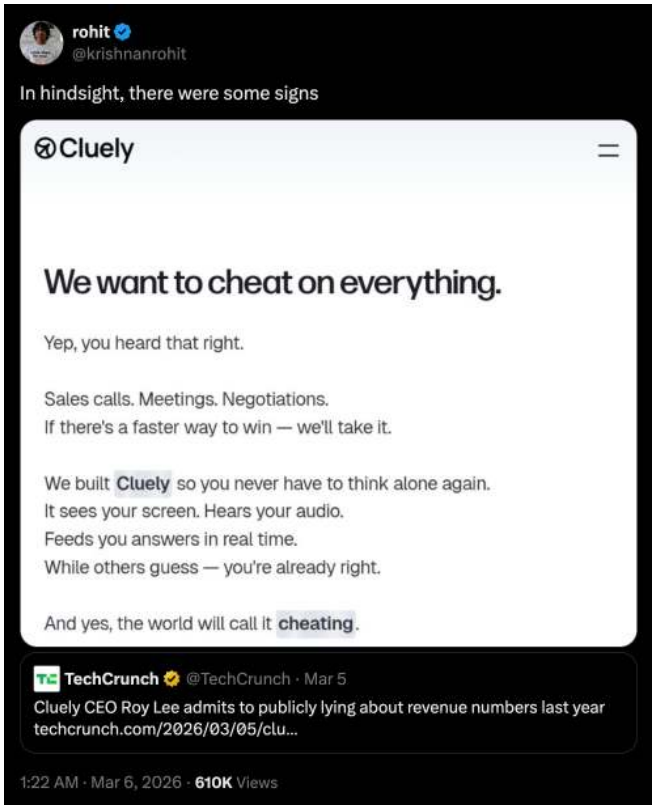
La vraie valeur ajoutée humaine n'a jamais été aussi clairement désignée : le *bon goût* et le *sens des responsabilités*, qualités dont sont précisément dépourvus ces "professionnels" qui nous noient dans des "idéologies" entières, tout comme ceux qui *vendent* ce à quoi ils ne comprennent rien.

Anti hype

Ce même décembre 2025, cette startup dont le nom sera tu était présentée dans certains médias comme "*le test IA No-Code qui ridiculise Selenium et fait trembler BrowserStack*". Rien que ça. Bonne chance à eux, l'entrepreneuriat est un monde de paris et d'aléatoire, où les meilleures idées arrivent au cimetière tout autant que les plus farfelues. L'analyse fondamentale s'y fait régulièrement dépasser.

Pour autant, rien de cela ne coupera Ocarina d'une vision qui se développe depuis plus de 15 ans, loin de l'arrogance de tous ces *enfants de la hype*.

Quoi qu'il advienne, nous leur souhaitons tout de même de s'éviter une humiliation aussi fulgurante que celle du founder de *Cluely* en cas d'échec et les invitons à reconsidérer leur arrogance.



Le CEO de Cluely, Roy Lee, admet avoir publiquement menti à propos des revenus de sa startup l'année passée. Krishman Rohit rappelle alors que depuis le tout début, leur proposition de valeur, c'était la fraude.

Aujourd'hui, le défi d'être le plus gros menteur devient de moins en moins rentable.
Et n'en soyons que bien-portants. De nouveau, avec l'IA, ce maillon qu'il manquait cruellement, crions-le, aussi fort que l'on criait "**HACK THE PLANET**" en 99 : **CODE IS LAW.**

Anti slipologues

Pendant trop longtemps, **l'informatique a été prise en otage** par une minorité, un "1%", qui a cru bon de la **transformer en terrain de jeux pour initiés** : "trucs de geeks", "techniques de ninjas", "orienté objet". **Le verdict est sans appel : ça ne tient pas, ou alors très mal.**

Ce que l'on pensait depuis les années 1930, **depuis l'invention du λ-calcul**, est enfin scalable à l'échelle que l'on aurait toujours voulu lui donner.
N'a-t-on jamais connu une si belle *formalisation* avec une telle paternité dans l'Histoire de notre industrie ?

Les récentes évolutions du *système de types* de Python, sur lequel Ocarina s'appuie profondément, font partie de la raison-même de sa faisabilité.
Sans non plus être "le futur", **puisqu'il s'agit d'une science établie mais qui est arrivée cruellement tardivement dans notre écosystème.**

Tout comme l'IA, qui achève tranquillement de rendre la capacité de nuisance de ce "1%" obsolète.

La gratitude que l'on doit à ces progrès technologiques est aussi immense que la colère de tous ceux qui pestent après, que nous surnommons dans nos contrées les *slipologues*.

Lire aussi : *Haters*, par *Paul Graham*.[6]

Pour toutes ces raisons, j'ai pris la décision de **ne JAMAIS perdre mon temps à argumenter lorsque ce n'est pas la peine.** J'apprécie argumenter avec des personnes **focalisées sur la résolution de problèmes et qui sont douées de bon sens.**

Pour le reste : **aucun regret à fermer toute PR ou issue inutile.**
Certaines personnes ne sont tout bonnement pas faites pour me plaire, et qu'il en soit ainsi.

Ocarina est *le premier de mes projets que j'ouvre au public*, **mais d'autres sont aussi à venir, dont des nettement plus intéressants.**

From the underground,
For Now and Forever.

Bonne continuation.[7]

"Un idiot admire la complexité, un génie admire la simplicité, un physicien essaie de simplifier. Pour un idiot, plus c'est compliqué, plus il est admiratif : si l'on produit quelque chose de tellement bordélique qu'il n'y comprend rien, il va te prendre pour un dieu parce que tu as fait quelque chose que personne ne peut comprendre. C'est comme ça qu'on écrit dans les revues académiques : on essaie de tout rendre le plus compliqué possible afin que les gens croient que ces torchons sont géniaux."

— Terry Davis, "le programmeur le plus intelligent qui ait jamais existé"

"Pour avoir de la connaissance, ajouter des choses chaque jour. Pour avoir de la sagesse, enlever des choses chaque jour."

— Lao-Tseu

---

[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

Chapitre 3

Premiers pas

On ne voyage pas pour arriver, mais pour voyager.

Avertissement

> ***Note :*** *Ce livre a pour but de faciliter la prise en main du projet* `ocarina-example` *fourni, qui reste* ***la source de vérité*** *à consulter en toutes circonstances.*

> ⚠️ *Le livre sacré d'Ocarina N'EST PAS et ne sera jamais "clé en main". Ocarina demande une certaine maturité pour être utilisé. Par conséquent, nous ne nous focaliserons que sur ce qui peut réellement être piégeux.*

Cette page explique le chemin avant toute chose. De la pratique sera nécessaire dans tous les cas.
📖 Munissez-vous de l'exemple canonique comme référence.[1]

1. Mise en place du projet

Créez un nouveau projet Python, puis installez les dépendances nécessaires :

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

Ensuite, créez votre structure de dossiers.

2. Les adapters

Ocarina repose sur un système d'*adapters* que l'utilisateur a la responsabilité d'écrire. Ils permettent de configurer le framework selon les contraintes et conventions propres à chaque projet.

Les *adapters* principaux à créer sont les suivants :

- `act` *(requis)*
- `test_campaign` *(requis)*
- `test_suite` *(requis)*
- `env_getters` *(facultatif)*
- `match_page` *(facultatif)*

2.1 EnvGetters

L'`EnvGetters` d'Ocarina centralise et type l'accès aux variables d'environnement. Il se divise en deux catégories :

- **Creds** : paires login/mot de passe, exprimées sous forme de dictionnaires immutables.
- **Values** : valeurs individuelles (chaînes de caractères).

```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)
```

Une fois cet *adapter* en place, il devient possible de récupérer une valeur ou des credentials de la façon suivante :

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

> ***Note :*** *Les clés valides sont à fournir à travers deux types tel que :* `EnvGetters[_CredsKeys, _ValuesKeys]`*. Dans le cas où l'utilisateur ne souhaite utiliser QUE la fonctionnalité* `.get_value()`*, il suffit de typer* `_CredsKeys` *tel que :* `Never`*. Il en va de même pour* `_ValuesKeys` *à typer en tant que* `Never` *si l'utilisateur ne souhaite utiliser QUE la fonctionnalité* `.get_credentials()`*.*

Nos accesseurs sont alors strictement typés, par exemple :

```
redis_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

Dans Ocarina, `act` est le verbe utilisé pour exprimer un pas de test au sein d'un scénario. Sa construction est intentionnellement laissée à la charge de l'utilisateur, pour des raisons abordées plus loin dans ce livre (*hooks*).

Sa forme minimale est la suivante :

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

    return create_act(
        pom,
        action,
    )
```

2.3 TestCampaign

L'*adapter* `TestCampaign` est volontairement minimaliste. La seule information qu'Ocarina ne peut pas deviner est le **nombre de workers**, c'est-à-dire le nombre de navigateurs à faire tourner en parallèle pour une campagne. Ce paramètre pouvant aussi être passé directement via la CLI, un petit *adapter* suffit :

```
@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,
        )
```

> *Le type du* `WebDriver` *utilisé (Selenium ou autre) est injecté ici :* `OriginalTestCampaign[WebDriver]`.
> *Et ici :* `suites: Sequence[TestSuite[WebDriver]]`

> ✅ *Bien évidemment, insérez VOTRE `TestSuite` adaptée ici, pas la built-in d'Ocarina.*

2.4 TestSuite

C'est l'*adapter* le plus important à comprendre. `TestSuite` expose nativement un grand nombre de paramètres. L'objectif de cet *adapter* est de créer une **façade** : certaines valeurs sont figées une bonne fois pour toutes (*hard-codées*), d'autres sont exposées optionnellement avec des valeurs par défaut. C'est un *rétrécissement*.

Par exemple :

```python
@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,
        )
```

> *Le type du `WebDriver` utilisé (Selenium ou autre) est injecté ici :*
> `OriginalTestSuite[WebDriver]`.
> *Ainsi qu'ici :* `tests: Sequence[Test[WebDriver]]`
> *Et ici :* `drivers_pool: SeleniumWebDriversPool`

Transient errors

La notion de `transient_errors` est centrale dans `TestSuite`.
Ces erreurs sont traitées comme du **bruit** : si un test échoue à cause d'une exception listée dans `transient_errors`, il est automatiquement rejoué.
Le nombre maximum de tentatives est défini par `max_retries_per_test`.

Ce mécanisme rend l'exécution des tests tolérante à la *flakiness*. Les tests qui rejouent fréquemment apparaissent clairement dans les logs, ce qui permet aux mainteneurs d'identifier et corriger les sources d'instabilité, qu'elles soient liées à une mauvaise utilisation de Selenium, à des conditions d'environnement hors de portée, ou à d'autres facteurs externes.

Only IDs et exclude IDs

Ces deux paramètres permettent l'exécution conditionnelle de tests.
Ce sont des filtres par ID.

> ⚠️ ***Attention à bien les inclure dans cet adapter, sinon ces valeurs passées depuis la CLI ne seront pas prises en compte.***

2.5 MatchPage

`match_page` est un opérateur d'Ocarina conçu pour gérer les pages à rendu non déterministe : bannières de cookies, challenges anti-bot, A/B tests, etc.

Son fonctionnement repose sur un principe simple : **toute exception levée est interprétée comme un non-match, et donc avalée par** `match_page`. Il est cependant possible d'exclure certaines exceptions de cette mécanique, afin qu'elles remontent normalement dans le flot d'exécution.

Par souci de cohérence, on souhaite généralement que les `transient_errors` soient dans ce cas : elles doivent remonter plutôt qu'être silencieusement avalées.

L'*adapter* se crée tel que :

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

3. Écrire un premier POM

Le pattern POM (*Page Object Model*) étant un standard bien établi, nous n'en reprenons pas la définition ici.

Voici comment créer son premier POM avec 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:
        """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
```

Quelques points méritent d'être détaillés.

3.1 SeleniumTitleMixin

Tout objet héritant de `POMBase` doit implémenter une méthode `get_current_title`. `SeleniumTitleMixin` fournit cette implémentation de façon transparente, sans qu'il soit nécessaire de l'écrire manuellement.

Son rôle ne s'arrête pas là : il définit également l'attribut `_driver` avec le type `WebDriver` (Selenium), ce qui le rend **incompatible avec tout autre type**. Tenter d'y assigner une valeur incorrecte produira immédiatement une erreur de typage :

```
self._driver = "lol"

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

`SeleniumTitleMixin` joue donc aussi un rôle de **détrompeur de typage**. Des mixins analogues existent ou peuvent être créés pour d'autres technologies d'automatisation de navigateur.

3.2 Retourner `self`

Chaque méthode d'action retourne `self`. C'est un choix de design volontaire dans Ocarina, à respecter systématiquement, il permet le chaînage des appels et la composition fluide des scénarios.

4. Écrire des connectors

Les connectors sont une couche fine mais indispensable pour la lisibilité des scénarios. Ils encapsulent les appels aux méthodes du POM derrière des fonctions nommées explicitement :

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


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

Il est également possible de les composer directement :

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

5. Écrire un premier scénario

Les briques sont en place.
Voici comment les assembler en scénario :

```
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)
    ),
)
```

Chaque pas de test est exprimé via `act`, auquel on chaîne un handler `.failure()` et un handler `.success()`.
Le scénario est ensuite encapsulé dans un objet `Test` via `create_selenium_test`.

6. Créer une suite de test

Une suite regroupe un ensemble de tests à exécuter sur une même *pool* de drivers :

```
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. Créer une campagne de test

Une campagne regroupe plusieurs 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. Créer un cycle de test

Un cycle regroupe plusieurs campagnes. C'est l'unité d'exécution de plus haut niveau :

```
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. Bootstrapper le projet

Voici le point d'entrée complet du projet :

```
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",)),
        ),
    )
```

Le déroulement est le suivant :

1. Les arguments récupérés via la CLI sont poussés dans un store global.
2. Une *pool* de drivers est créée, c'est celle-ci qui gère le cycle de vie des navigateurs web en parallèle.
3. Une callback `_post_exec` est définie : elle s'exécute après les tests et les plugins, affiche les résultats, et retourne un code d'erreur si le cycle a échoué.
4. L'ensemble est bootstrappé à l'intérieur d'un chronomètre mesurant la durée totale d'exécution. Le flot d'exécution est donc : **cycle → plugins → post_exec**.

> ℹ️ *Les plugins sont des fonctions déférées passées à* `run_plugins`.
> `run_plugins` *prend* `results` *en argument,*
> *ce qui indique sans ambiguïté par simple lecture de signature de fonction qu'ils s'exécutent en post-traitement, une fois les résultats disponibles.*

---

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

Chapitre 4

Premiers scénarios

La langue n'est ni réactionnaire, ni progressiste. Elle est tout simplement fasciste, car le fascisme, ce n'est pas d'empêcher de dire, c'est d'obliger à dire.

La fin des états invalides

... Rendre les états invalides impossibles à représenter.

Nous allons détailler le fonctionnement d'`act`, de `drive_page` et de `match_page` dans l'écriture des scénarios de test avec Ocarina.

Act et drive_page

Exemple canonique

Commençons par un exemple que nous allons progressivement casser :

```
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` exprime que l'on prend le contrôle d'*une* page.
Toute *transition* devient explicite par l'ouverture d'un nouveau `drive_page`.
À l'intérieur, `act` exprime une action émise sur cette page : c'est un *pas de test*. `drive_page` est variadique, elle accepte autant de `act` que nécessaire, et la virgule entre chacun devient un ET :

> *Ouvre, puis vérifie la page d'accueil. ET clique sur le CTA. On change de page : vérifie que l'on est sur la page pour réserver un appel.*

Système immunitaire

Essayons d'appeler `verify_book_call_page` sur `homepage` :

```
act(on_homepage, verify_book_call_page)

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

L'action est incompatible avec sa cible. Ce programme ne *compile* pas.

Oublions un `.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
```

Plaçons un `.success` immédiatement après `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'
```

Inversons `.success` et `.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'
```

Chaînons des `act` hétérogènes dans un même `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
```

AUCUNE de ces fantaisies ne compile. Ocarina est anti *petit malin*.

Beaucoup se battent avec des *coding styles*.
Ocarina est plus tranché : si le style n'est pas respecté, ce n'est pas une erreur de *lint*, ce n'est pas un avertissement. Ça ne **compile** pas.

Ocarina force le même gabarit pour tout le monde, et ces erreurs apparaissent directement dans l'éditeur via *mypy* : feedback instantané.

La priorité des scénarios de test est leur uniformité et leur simplicité. C'est tout.

match_page

`match_page` gère les situations où une page peut être rendue de manière différente.

Commençons par le principe des *matchers* :

```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
```

Un *matcher* vérifie de manière minimale si quelque chose est vrai, en allant au plus vite.

> ⚠️ *Il vaut mieux éviter un `.find_element(s)` brut : c'est la voie rapide vers la flakiness.*

Le délai maximal de 5 secondes n'aura aucun impact dans une batterie scalée horizontalement, ce n'est donc pas une pratique à craindre ici. Il n'est pas non plus recommandé de déguiser un `verify` en *matcher* : **ce sont deux outils différents.**

Usage dans un scénario :

```
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` se pose au même niveau que `drive_page` et est chaînable. Sa commande `then` attend à nouveau une chaîne de `drive_page` ou `match_page`. Les branches sont définies par `when`.

`match_page` et `when` ont été ajoutés après coup, l'Igoristan était tellement aléatoire que le cas d'usage s'est imposé de lui-même.
Leur implémentation a été simple, preuve de la flexibilité de la grammaire : d'autres structures analogues pourraient très bien suivre.

Répétitions

Pour répéter une chaîne de test (par exemple, pour tester plusieurs tentatives d'accès non autorisé), il suffit de multiplier la liste :

```
[
    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

Un *fragment* est une fonction `(driver, logger) -> TestChain` injectable avant ou après la chaîne principale, via `pre_test_scenarios_fragments` et `post_test_scenarios_fragments`.

Par exemple, `login_without_otp_happy_path` est un 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!")),
            # * ...
        ),
        # * ...
    ]
```

Injection au début :

```
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],  # <- [!]
)
```

Injection à la fin :

```
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],  # <- [!]
)
```

Les deux paramètres peuvent être combinés et acceptent chacun une liste de *fragments*, injectés dans l'ordre fourni.

Aliasing

Les scénarios peuvent devenir lourds.
Comme tout y est déclaratif, l'utilisateur est libre de créer des alias :

```
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(...)),
]
```

Toute valeur peut être aliasée et réutilisée.
Cette écriture est pure, elle ne provoque aucun *effet* immédiat.
Tout peut être redéclaré ailleurs, réorganisé ailleurs, tant que la chaîne finale correspond à l'attendu.

Chapitre 5

Premiers jutsus

Vois, sous la glace, le ruisseau brille…

Jeux de données

Piloter un test grâce à un jeu de données est très simple avec 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
]
```

Une *closure*[1] suffit.
On notera que `Scenario` est déclaré depuis l'intérieur ici : logique, puisqu'il s'agit de l'encapsuler.

`multi_login_tests` est donc une *liste* de `Test`, que l'on *unpack* dans une `TestSuite`, tel que :

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

```
        drivers_pool=drivers_pool,
    )
```

Tests de fumée

Pour lancer des tests de fumée en début de cycle avec 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` accepte deux valeurs (par défaut, `"fail-fast-on-first-smoke-campaigns-sequence-fail"`) :

- `"fail-fast-on-first-smoke-campaigns-sequence-fail"` : dès qu'une campagne de tests de fumée échoue, les suivantes sont passées (*skip*).
- `"wait-for-all-smoke-tests"` : toutes les campagnes de tests de fumée s'exécutent, même en cas d'échec intermédiaire.

Dans les deux cas, les tests principaux sont ignorés si au moins un test de fumée a échoué.

Setup et teardown

`Scenario` accepte deux *callbacks* optionnelles : `setup` et `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,
)
```

Cycle de vie

1. `setup()` : exécuté avant la `test_chain`. En cas d'échec, la `test_chain` est ignorée et le `teardown` s'exécute quand même. Si toutes les tentatives échouent à cause du setup, le test est marqué **SKIPPED** (pas **FAILED**),
2. `test_chain` : les étapes du test,
3. `teardown()` : toujours exécuté, même en cas d'échec. Les erreurs sont loggées et ignorées.

`setup` et `teardown` sont des `Effect`.
Elles sont destinées aux préoccupations d'infrastructure : seeder une base de données, appeler une API, nettoyer un état…

Si une encapsulation est nécessaire : *closure*.

Proxy pattern

Un cas d'usage de l'exemple canonique[2] est `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)
```

Le principe est de retourner des *Web Elements* qui intègrent des comportements utilisateurs à l'interaction : les frappes clavier, en l'occurrence.
Transparent pour le *système de types*, transparent pour le *runtime*.

On peut alors faire :

```
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,
        ),
    ),
)
```

Ou, avec une *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)  # <- [!]
```

Le même principe s'applique au logger, pour le router vers un *sink*, par exemple, bien que ce cas ne soit pas couvert canoniquement par Ocarina.

Programmation réactive : NON

Les scénarios de test d'Ocarina sont volontairement statiques.
Pourtant, une application web est dynamique et parfois, enregistrer une valeur à la volée pour la passer à une étape suivante est tout à fait légitime.

Ocarina n'y répond pas. Il n'en a pas besoin.

Réponse architecturale

Ce qu'on cherche ici est un *cache in-memory*.

On génère des clés juste avant le lancement de la chaîne de test, et on les passe aux actions du POM. Les actions enregistrent et consomment via une clé unique.
Le scénario se contente de les fournir :

```
# * ...
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!")),
    ),
    # * ...
]
```

Appels API et verrous

Les appels API et les verrous sont à gérer dans les POMs.

> ⚠️ *Ocarina ne supporte pas `async`/`await` et ne le fera pas.*

Appels API : `requests` (synchrone) suffit.
Verrous : `threading.Lock` si un seul process à la fois, sinon verrous distribués Redis (`redis.StrictRedis` + `redis.lock`).

Profil navigateur

Certains cas nécessitent de passer un profil via `--profile-path` :

- **Authentification proxy**,
- **Extensions préchargées**,
- **Paramètres locaux** (langue, timezone, certificats…),
- Etc.

---

[1] https://fr.wikipedia.org/wiki/Fermeture_(informatique)
[2] https://github.com/mojo-molotov/ocarina-example

Chapitre 6

Premiers obstacles du monde réel

La seule attitude judicieuse consiste à s'accommoder de l'état des choses.

Aléas serveur

Les erreurs serveur aléatoires sont coriaces.
Durant une expérience de travail particulièrement pénible, j'ai eu à faire face à un environnement qui affichait régulièrement des pages d'erreur 500 totalement aléatoires, quelle que soit la zone explorée de l'application.

Dans ce genre de cas, Ocarina propose une réponse directement à la création du verbe `act` :

```
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,  # <- [!]
    )
```

Le hook `on_failure` a précisément été conçu pour ça.
Il suffit de créer des *guards* et de modifier l'erreur encapsulée dans `Fail` pour provoquer un *rejeu* du test ayant échoué en raison d'une cause externe.

L'étape suivante devrait paraître familière :

```
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,
        )
```

Enfin, si `match_page` est également utilisé dans le projet et qu'une variable partagée `transient_errors` n'est pas souhaitée, il ne faudra pas oublier d'ajouter ces nouvelles définitions d'erreurs aléatoires dans le `raised_exceptions` du constructeur de `match_page`.

Les logs des tests automatisés permettront d'amorcer une première conversation autour de ces problèmes.

Aléas de pas de test

Pensant avoir laissé derrière moi ce genre de désagréments, j'ai changé de crémerie… pour y découvrir des formulaires instables et des systèmes d'authentification qui fonctionnaient une fois sur deux.

Face à cela, la réponse d'Ocarina est différente : on délègue la responsabilité au 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()
            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
```

Cela soulève aussi une question sur les *connectors* : comment leur ajouter des paramètres ?

```
"""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
```

Il suffit de retourner le `def` avec la signature attendue, à l'intérieur d'une fonction qui capture les paramètres.
C'est une *closure*[1].

Aléas de Selenium

Selenium offre de nombreuses occasions de se tirer une balle dans le pied : *race conditions*, erreurs de "*stale element*", etc.

La réponse ici est **pragmatique** : ajouter `WebDriverException` directement aux `transient_errors`, avec un nombre de rejeux généreux (8, soit 9 vies, comme un chat 🐱).

On capture toutes les erreurs Selenium et on observe les rejeux dans les logs.
De là, il devient possible d'identifier les tests qui mériteraient d'être améliorés.

Erreurs aléatoires discrètes

Plus surprenant encore : des applications affichant des toasts d'erreur sans raison apparente, ou des formulaires signalant des erreurs de validation sur des saisies pourtant correctes, sans pour autant bloquer le parcours.

Ces erreurs sont les plus pénibles à détecter, car elles sont *indolores*. On ne peut pas simplement constater un crash et ajouter une *politique de retry* en attendant que l'anomalie soit corrigée. Elles sont, pour ainsi dire, invisibles.

Il resterait à massacrer les scénarios de test ou à recourir à des "techniques de ninjas".
Ocarina refuse ces deux options.

La solution, les *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` est la *callback* que le *watcher* appellera toutes les 0.8 secondes (`poll_interval`).

Détaillons un peu plus l'approche.

Utilisation de Javascript

Le *watcher* est tolérant aux erreurs : il les avale silencieusement.
Il n'y a donc aucun intérêt à utiliser une fonction Selenium pour capturer un élément de la page, si ce n'est s'encombrer.
Utiliser les fonctions natives de Selenium imposerait de gérer des questions d'*implicit timeout*.

Passer directement par du *Javascript* permet de contourner toute logique de *polling* interne et de rendre l'exécution du *watcher* la moins bloquante possible pour le test qui tourne sur le même *driver*.

Ce tour de passe-passe devient alors invisible, puisqu'il n'est que d'une durée de quelques millisecondes.

Fingerprinting

Les *watchers* exposent un cache simple (de strings), pensé spécifiquement pour ce besoin : si la même erreur reste visible et est détectée toutes les 0,8 secondes, inutile de la voir apparaître plusieurs fois dans les captures d'écran, les rapports et les logs. Le fingerprint permet d'ignorer ce qu'on a déjà vu.

Report

La *callback* finit par : `watcher.report`.
Cet appel s'occupe de :

1. Logguer la friction détectée par le watcher,
2. Prendre une capture d'écran comme trace de ce qui a été détecté.

HumanizedDriver

Rien ne nous empêche de greffer des comportements sur le *logger* ou sur le *driver*. Ici, le formulaire étant capricieux, on opte pour un test lent et "humanisé" : saisie avec fautes de frappe, corrections, hésitations. On wrappe simplement le *driver* dans un *proxy*, `HumanizedDriver`.

Heisenbugs de concurrence

Ma quête n'était alors pas terminée.
J'ai vu des collègues compter de 1 à 3 avant de tous cliquer en même temps pour émettre une même action dans l'*open space*. Je me suis alors questionné sur le sens de ma vie. En procédant de telle sorte, ils ont pourtant vraiment réussi à provoquer des anomalies.

Ce comportement peut être reproduit par Ocarina.
Par défaut, Ocarina est agressif.

Son option `saturate_workers` permet de forcer du clonage aléatoire de tests à l'intérieur d'une suite.

Dès lors qu'il y a plus de *workers* disponibles dans la *DriversPool* que de tests à exécuter dans une suite, Ocarina va alors cloner les tests aléatoirement, démarrer tous les drivers, et tous leur assigner un test à effectuer.

Il est possible d'activer cette option depuis la fonction `bootstrap`.
Il est aussi possible de l'activer ou de la désactiver individuellement, soit au niveau d'une suite, soit au niveau d'une campagne. En cas de contradiction, c'est l'élément le plus profond de

l'arborescence qui a le dernier mot. Par exemple, si une campagne dit de désactiver l'option, mais qu'une suite dit de l'activer, alors la suite prend la priorité.

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


# * ...
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.
        # * ...
    )
```

Il est possible de temporairement créer une suite avec seulement un test pour maximiser le ciblage. Ou encore de lancer sur plusieurs machines à la fois le cycle (scaling horizontal).

Au-delà des problématiques de concurrence, ce mécanisme de clonage vise également à s'assurer que les tests en succès ne doivent rien au hasard. Cet effet est amplifié par le degré de scaling horizontal et le nombre de workers impliqués.

---

[1] https://fr.wikipedia.org/wiki/Fermeture_(informatique)

Chapitre 7

Extensibilité

Si l'on ne croit pas à la liberté d'expression pour les gens qu'on méprise, on n'y croit pas du tout.

ValidationChain

Utilisable dans les POMs, `validate` permet d'exprimer des invariants sous forme de chaînes. L'exécution est **différée** : il faut appeler `.execute()` explicitement.

Le résultat expose `is_valid`, `errors` et `validated_values`. Il est inerte par défaut. `.raise_if_invalid()` remonte l'exception si besoin.

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

Chaînage d'invariants

Plusieurs assertions sur une même valeur :

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

Chaînage de validations

Plusieurs validations sur des valeurs différentes :

```
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()
```

Invariants réutilisables

Pour factoriser une validation récurrente, créer un *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` pour les invariants techniques, `BusinessInvariantValidator.create` pour le métier.

Assertions personnalisées

Sans argument :

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

Avec 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

Le *type checker* détecte les assertions incompatibles avec le type de la valeur :

```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 et failure

`.success` et `.failure` prennent chacun un *effet* à exécuter.
L'exemple canonique[1] implémente plusieurs handlers : log simple d'erreur, log avec URL courante, log de succès, et log de succès avec 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
```

```
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
```

D'autres *handlers* sont envisageables :

- `create_log_error_with_retry_hint` : signale une *transient error* et donc la possibilité de flakiness,
- `create_log_error_and_send_alert` : envoie un webhook à l'échec, sans polluer le test,
- `create_log_success_and_record_timing` : capture un timestamp de fin pour mesurer la durée d'un step (à combiner avec `on_run_effect` de `create_act`),
- Etc.

La création d'un *combinateur* est également envisageable.

Plugins

`bootstrap` permet de lancer des plugins post-exécution basés sur les résultats du cycle de test. Par exemple, `generate_docx_proof` parcourt l'arborescence de logs et génère un document Word (preuve de test) par cas de test, en insérant les captures d'écran et en convertissant les dates UTC en heure locale.

Le principe : les plugins réassemblent les artefacts générés en cours de route sous une forme différente. Un plugin pour créer un rapport sous forme de tableau de bord web serait par exemple tout à fait envisageable.

Grammaire extensible

La grammaire des scénarios de test repose sur un seul type : `ChainRunner[T]`. Un scénario est une `list[ChainRunner]` exécutée séquentiellement, court-circuitée au premier échec. `drive_page` n'est qu'une fine enveloppe autour de `chain_actions`, qui construit un `ChainRunner`. N'importe quelle fonction renvoyant un `ChainRunner` s'insère sans toucher au framework.

`match_page` a été ajouté après coup pour gérer les pages à état variable (banners optionnels, A/B tests, pages de maintenance…) : elle évalue des conditions dans l'ordre et exécute la première branche correspondante.

Autre exemple envisageable : `skip_if`, qui court-circuiterait volontairement une portion du scénario sur une condition sans échouer (retournerait un `Ok` neutre), utile pour des étapes optionnelles selon l'environnement ou les données de test.

La seule contrainte du point d'extension : retourner un `ChainRunner`.

---

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

Chapitre 8

Utiliser Ocarina avec l'IA

Hello it's me, your new best friend!

Un setup de travail : un cycle de test complet construit avec Claude Code et Ocarina, contre la démo publique Katalon CURA.

📖 Munissez-vous de l'exemple avec IA comme référence.[1]

Les trois pierres ancestrales

1. `CLAUDE.md` à la racine du projet.
2. `skills/` avec un `<nom>/SKILL.md` par procédure.
3. Règle de vérification : toute affirmation sur le SUT vient d'une observation (sonde, `gh api`, `curl -v`), jamais d'une inférence.

`CLAUDE.md`

Deux variantes. `CLAUDE.md` est complet (règles + organisation du projet, hiérarchie, conventions, forme CI, gabarit de PR). `CLAUDE.slim.md` ne contient que les règles. Slim quand le contexte est chargé ; complet pour l'onboarding et les revues. En cas de divergence, le complet l'emporte.

Les étapes d'onboarding (venv, `pip install`, la batterie de skills copiée dans Claude Code, `ruff` / `mypy` / `pre-commit`, smoke-check du runner) vivent dans `setup-environment`.

Les règles :

Les tests de sécurité sont fonctionnels et statiques, jamais actifs. Pas de payloads, pas de requêtes forgées, pas de manipulation du DOM via DevTools. Les scénarios black-hat passent par une UI normale.

Utiliser des constantes. Les valeurs nommées ne sont pas inlinées.

Les datasets sont des décisions humaines. Proposer, PAS exécuter.

Vérifier empiriquement le comportement du SUT. Sonde, `gh api`, ou `curl -v`. Jamais d'inférence. Re-dériver à chaque fois : une sonde ne répond que pour ce qu'elle a exécuté ; un diagnostic antérieur ne répond que pour cette exécution-là.

Chaque règle contient un "*pourquoi*" d'une ligne.

`skills/`

Un fichier Markdown par skill, frontmatter YAML + corps. Dix familles.

Review (13)

Lectures statiques ; remontent des constats.

- `review-spec-gaps` — questions de clarification sur les SFD.
- `review-watcher-misuse` — `watcher.report(...)` principe de « négatif uniquement ».

- `review-compartmentalisation-leaks` — URLs, sélecteurs, nombres magiques aux mauvais endroits.
- `review-dead-code` — connecteurs / POMs / scénarios / suites / fragments / constantes non utilisés ; au cas par cas : supprimer, mettre en incubateur (`<racine-source>/incubator/`, arbre de dépendances préservé), ou conserver.
- `review-report` — classifie chaque FAIL / SKIP d'une exécution.
- Et : `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` — élargit le filet des erreurs transitoires ; les morts chroniques sont de vraies flakes.
- `analyse-fixture-flakiness` — instrumente setup/teardown ; rend visibles les contaminations entre tests.
- `analyse-watcher-flakiness` — analyse la fiabilité des watchers.
- `analyse-screenshot-flakiness` — regroupe par `(test, étape, navigateur)`, détecte les différences.

Black-hat (6)

- `business-attack-ideation` — faire tomber le produit.
- `incoherence-attack-ideation` — chaque étape légale prise isolément, incohérent quand combinées pour construire un ensemble invalide.
- `persistence-attack-ideation` — tentatives répétées sur une action bloquée.
- `permission-appropriateness-audit` — le modèle d'accès est-il lui-même approprié ?
- `bfcache-exposure-ideation` — attaques BFCache.
- `lateral-resource-ideation` — IDOR via la barre d'adresse uniquement.

Comprehend (4)

- `assess-test-base` — catalogue la base de test.
- `assess-ecosystem` — recherche publique bornée, plafonnée par budget de tokens.
- `understand-sut-constraints` — bornes SUT qui cassent les tests parallèles.
- `understand-ocarina` — parcourt la doc.

Pick (3)

Par mtime, jamais par nom de fichier.

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

Author (8)

Chacun produit un livrable.

- `empiricism` — vérifier avant d'encoder ; ne pas écraser un test gap en échec intentionnel.

- `write-a-probe` — script jetable, gitignored.
- `write-test-strategy` — génère le document de stratégie de test à partir de la suite (scope, types, tables de couverture, arbre du cycle, pass/fail, gaps, matrice CI).
- `extend-coverage` — étend la couverture à partir du patrimoine existant.
- `update-frd-and-tests` — propage une mise à jour de spec.
- `manual-reproduction-guide` — repro exécutable par un humain.
- `manage-backlog` — `BACKLOG.md`.
- `pr-report` — rapport de PR adapté au type.

Refactor (2)

- `refactor-fragmentation` — DRY selon préférence utilisateur.
- `introduce-pom-retries` — retries internes aux POMs, avec dédoublement (first-try + with-retries).

State (1)

- `question-state` — interroger l'environnement avant de croire un résultat.

Setup (1)

- `setup-environment` — venv, outillage de dev, la batterie de skills Ocarina copiée dans le répertoire de skills de Claude Code, chemins de drivers dans `CLAUDE.local.md`, boucle pré-commit, smoke-check du runner.

Run (1)

- `propose-visual-review` — avant un lancement local, propose `--not-headless` (regarder le navigateur exécuter) vs headless (comme en CI). Compose la commande ; l'utilisateur la lance.

Chaînes récurrentes

Cycle en échec : `review-report` → `analyse-*` → `write-a-probe` → trouvailles propagées dans `IDENTIFIED_GAPS.md` / les SFD / un commentaire de scénario → sonde supprimée.

Scénario black-hat prometteur : `empiricism` → `extend-coverage` (souvent en échec intentionnel).

Changement de spec : `update-frd-and-tests` (SFD d'abord, tests ensuite). Les tests gap sont reformulés, pas basculés.

Nouvelle primitive Ocarina : `understand-ocarina` d'abord, écriture ensuite.

Lorsque l'on est sur le point de lancer une exécution : `propose-visual-review` — headed (`--not-headless`) ou headless (comme en CI) ? Compose la commande ; l'utilisateur la lance.

Discipline

Remonter, ne pas appliquer. Les skills produisent ; l'utilisateur décide.

Empirique plutôt qu'assertif. Toute affirmation SUT est observée, citée, datée. Phrase rituelle : *« Juste remarque, je suppose. Je vérifie empiriquement. »*

Les tests gap sont reformulés, pas basculés au vert. Inverser l'assertion, renommer, déplacer la ligne dans le doc de stratégie, consigner la date dans `IDENTIFIED_GAPS.md`. Le tout via `update-frd-and-tests`.

Les signaux des watchers sont toujours négatifs. Un watcher qui émet *« login réussi »* casse le contrat.

Distribué quand une ressource est partagée. Dès que plusieurs workers se partagent une ressource plafonnée par le SUT (sessions, créneaux, quotas), la coordination passe par des primitives distribuées. Sinon, un cache local en mémoire suffit — à condition que les clés soient garanties uniques et que leur génération soit thread-safe.

Mtime, pas nom de fichier. Les suffixes UUID sont aléatoires ; `pick-*` trie par mtime.

Ce que ce setup n'est pas

- Ne génère pas de tests de façon autonome.
- Ne patche pas les hallucinations en CI ; un échec déclenche `review-report` + `analyse-*`.
- Ne réécrit pas la spec ; seul `update-frd-and-tests` le fait, avec une ligne de révision.
- Ne fait pas de tests de sécurité actifs. Jamais.

Ressources exposées

- 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