Священная книга Ocarina

Игорь Казанова

Содержание

Глава 1

Что такое Ocarina?

Все платформы для тестирования сделали одну и ту же ошибку. Ocarina нет. Узнайте почему.

Ocarina была разработана, чтобы сделать **автоматизированные тесты на основе браузера максимально простыми**, одновременно давая пользователю **полный контроль**.

Еще одна!

Остальные

Большинство фреймворков для тестирования были созданы в мире, где граница между "теми, кто пишет код" и "теми, кто определяет тесты" была реальной и структурной.

Robot Framework попытался **обойти это** с помощью *DSL* (*предметно-ориентированного языка*), с **собственным форматом** и **собственной экосистемой плагинов**. В результате RF навязывает свои стандарты по умолчанию: это непосредственная цена его обещания.

Cucumber попытался то же самое с *Gherkin*: "естественным" языком, который на практике **ограничивает всех, не освобождая никого по-настоящему**. Стоимость: **постоянный слой перевода, рассинхронизация Gherkin/кода**.

Все они сделали одну ставку: **скрыть сложность**, чтобы преодолеть разрыв между профилями. Результат: нетехнические люди остаются зрителями, а технические люди оказываются **в ловушке инструмента, который они никогда не выбирали бы**.

Самая большая цена — это **гонка на дно: меньше опций, и "гибкость", которая может быть достигнута только *противодействием* вашим *инструментам*, а не использованием *решений***.

Нет пользы в том, чтобы тянуть веревку, которую нам дали, если то, что находится в центре, это просто гордиев узел.

Мы

Лучше одному, чем в плохой компании.

Ocarina делает противоположную ставку: эта граница исчезнет. Это не предмет дискуссии, вокруг которого все завязали петлю ненужно усложненными инструментами, принимая их за решения. Влияние: **операционная катастрофа** в момент, когда возникает потребность, которую нельзя выразить в рамках "фреймворка", который НЕ является по-настоящему универсальным.

И самое плохое: **все эти технологии будут продолжать эволюционировать в том же направлении**. НИ ОДНА из них не сделает этот *сдвиг*, потому что это потребует смены парадигмы, **возврата к основам, которые прямо противоречат всему их ценностному предложению**.

Однако остается потребность в коде тестов, который **читаемый, трассируемый и гибкий** в его наиболее **сыром** виде.

С AI и такими инструментами, как *Claude Code*, эта ставка укрепляется с каждым днем. Мост между техническими и нетехническими людьми больше не является слоем абстракции.

Это сам AI. AI, работающий с сырыми данными.

Культурная проблема

Есть еще одна слепая зона, редко названная: **методология**.

ISTQB, где ты?

ISTQB и профессиональные тестировщики потратили десятилетия на создание точного, проверенного боевым опытом словаря: *циклы тестирования*, *кампании*, *наборы тестов*, *тестовые случаи*, *шаги тестирования*. Четкая иерархия, разработанная для **организации, отслеживания и контроля** качества ПО.

Автоматизированные инструменты **в значительной степени игнорировали** это наследие.

pytest, *Jest*, *Mocha*... все это **гибридные смеси**, где тестировщики должны учиться думать как разработчики, и где никто по-настоящему не говорит на одном языке.

Этот "двуязычный метод" — это неудача.

Возврат к основам

Ocarina не делает такие компромиссы.

Её структура моделируется **напрямую и исключительно** на основе методологии тестировщика. Каждое понятие в коде соответствует концепции в домене тестирования. Без заимствований, без переназначения, никаких "приблизительно".

И потому что Ocarina придерживается этой ставки полностью: **она полностью автономна**. Нет плагина pytest. Нет вынужденной интеграции в экосистему третьей стороны. Ocarina это *batteries-included*, ей ничего больше не нужно для работы, и это сознательный выбор.

Реальная проблема

О "творчестве"

Все одержимы *как*: интерфейсы, абстракции, "красивые" DSL.
Между тем, *почему* исчезает.

Ocarina делает противоположный выбор.

Вся её конструкция и эта Святая Книга сосредоточены на **почему**: на реальных проблемах.
Не на абстракции для использования *"как вам угодно"*.

А как же *как*?
Ответ простой: Ocarina **ёмкая, сразу работоспособная и строгая**. Построена так, чтобы и люди, и LLM могли понять её ядро и использование без трений.

Какая наука?

Здесь всё основано на **статических основах**:

- типы,
- дженерики,
- функциональное программирование.

Ocarina затрудняет неправильное использование по конструкции: компилятор восторжествует.

Просто алгебра.

> 📖 *Алгебра происходит от арабского* الجبر *(al-jabr), что означает "воссоединение сломанных частей" или "исправление переломов".*

Большинство инструментов начинают с человеческой грамматики, чтобы якобы "формализовать".
Затем наступает осознание, что машина не может это читать как есть.
Итак, давайте наваливать "адаптеры".

Мы не называем это *формализацией*, но *пустым желанием*.

Ocarina, очевидно, делает **противоположное**.
Вот что делает Ocarina **стабильной и одновременно расширяемой**.

Видение

Суверенная грамматика

Что, если делегирование вашей грамматики "стандартам" никогда не было хорошей идеей в первую очередь?

Реальный разрыв с E2E тестированием **не в навязывании "лучшего" Новояза**.
Короткий ответ: Ocarina **расширяема**. Любой глагол и союз могут быть созданы, все управляется **строгими правилами, которые сохраняют всё глубоко согласованным**.

Что остается, так это гарантировать **отслеживаемость и надежность**.

Рождённая практикой

В Ocarina каждый шаг наблюдаем.
Путь ошибки явный. Отчет тестирования **естественно вытекает из кода**.

Без переусложнения. Без ненужных зависимостей.
Просто что-то **маленькое, читаемое и построенное на века**.

Самое мощное то, что **Ocarina ничего не изобретает: Ocarina возвращается к основам**.

Принятие

В самых крайних случаях: Ocarina не нужно устанавливать.
Скопируй, адаптируй, запусти.
Нет зависимостей для проверки.

Для команд, ограниченных *политиками безопасности*: код маленький, проверяемый за полдня. Ничего скрытого. Единственные внешние зависимости находятся в плагинах после выполнения, и если один из них не подходит, его можно удалить без разрушения всего остального.

На практике консультант может появиться на месте клиента с Ocarina в кармане, *почти* без чьего-либо разрешения.

Устают не люди — их изнашивают отжившие струны.

И именно поэтому она существует: чтобы вернуть тестировщикам их **независимость**.
Всё это в **драгоценности синтеза**.

Глава 2

Первые отзывы

Первая критика Ocarina. Угадайте, какая.

С самого начала, Ocarina была спокойно представлена профессионалам со всего мира, из всех слоёв.

Первые жалобы

Первые "критики" часто были одинаковыми: *"Вы хвастаетесь чем-то, что очень простое."*

Некоторые остановились там.
Другие пошли дальше: пытались объяснить, что такое "настоящая сложность" по их мнению.

Но **ни один из них не смог бы достичь того же самого.**

Самый забавный момент в том, что **мне даже не пришлось поднимать сложность самому.** Я просто **представил несколько тестовых сценариев, написанных с Ocarina**, и **эта реакция появилась сразу же**.

Это просто результат оперантного обусловливания 99% людей в нашей отрасли.

Добавление еще одной передачи к непонятной машине кажется фетишем многих инженеров моджо.

Для них открытие капота было достаточным.

Во-первых: это не о хвастовстве.
Ocarina **не является продуктом шоумэнства**, в отличие от того, что тенденция многих

"инженеров" производить. Это просто продукт того, что *естественным образом возникает*.

Нарциссизм выражается через усложнение всего чрезвычайно ради *доминации*. Не через приобретение навыков и упрощение процессов, поскольку конечный результат — полный беспорядок: хаос.

Это также один из способов быстро заметить нарциссов в *казино*: чем больше машина мигает, чем "сложнее" она кажется, тем больше нарцисс захочет сесть и доказать, что они "умнее машины".

Их также можно узнать по их *непереносимости неудачи*. Они систематически хотят показать, как много ОНИ "подумали обо всём".

Никогда красного с нарциссом, только зелёного, только "готово к отправке в производство".

Последствия драматичны и радикально противоположны менталитету тестирования ПО: нарциссизм не приносит ничего хорошего в этом домене.

В любом случае, он приносит абсолютно ничего хорошего нигде: он приносит только *ад на земле*.

Ocarina радикально противоположна этим явлениям.

Демонстративная сложность

Настоящая сложность не хвастается. Она ощущается. В стабильности, в расширяемости, в том, что не ломается.

Нет добавленной стоимости в создании чего-то ужасно сложного в использовании, кроме как доказать, что это сложно. Никто это не просит.

Наша индустрия имеет серьёзную проблему с тем, что сложность рассматривается как знак серьёзности.

Эта сложность **существует исключительно для "впечатления коллег"** и не достигает ничего, кроме **переигрывания глупости**.
Отличие в неправильном направлении хуже, чем простая посредственность и это единственное "достижение" в этом: поздравляем, вы успешно достигли вершины **пирамиды бреда**, где больше нет места сомнению в себе!

Это также причина, по которой **KISS** (*Keep it simple, stupid*) так неправильно понимается в индустрии. Многие предполагают, что "простое" означает "неизощренное". Было бы трудно неправильнее прочитать принцип.
Поэтому мы в итоге получаем худшее из обоих миров.

Как можно претендовать на *продуктивность* таким образом? Серьезно?

Вы. Не. Даже. Знаете. Что. "Чистый. Код". Действительно. IS.
Вы даже не пробовали!

Философия

KISS

KISS находится в сердце Ocarina.

И если при чтении первого тестового сценария возникает реакция *"это супер просто..."* — было бы жаль воспринимать её как критику: это именно тот комплимент, который ищется.

Спасибо.

Чтобы добраться туда, вопрос никогда не был о "сиянии" ярче, чем кто-то другой.
Это никогда не было реальным вызовом с самого начала.
Вопрос был просто: **что мне действительно нужно?**

Некоторые люди имели другие идеи, довольно "творческие" из них:

- Искажение реализации ROP (*Railway Oriented Programming*) в Ocarina до потери всякого смысла, при том что они даже не знали, что такое ROP,
- Впихивание "*hooks*" и прочих так называемых "*ninja techniques*" прямо в середину тестовых шагов,
- Отказ от типизации,
- Переписание на Rust ради "производительности",
- Навязывание мне их неграмотного взгляда на ленивые вычисления и *IoC*, как если бы это было евангелием,
- "Объяснение" мне императивного против декларативного программирования, в то время как извергается полный бред,
- "Разговор" со мной о *событийно-ориентированном программировании*, в то время как продолжается бред,
- И в худшем случае — разглагольствования об ужасающей теории "декларативного объектно-ориентированного программирования",
- И т. д.

Даже кто-то, кого я раньше очень уважал, начал мне надоедать.
Он решил, что может меня чему-то научить, ведя себя нахально, говоря мне в неприятном тоне, "чего не хватает", когда всё, о чём он говорил, было не только в плане, но и намеренно скрыто по стратегическим причинам, но пошло бы намного дальше, чем что-либо, что он когда-либо мог себе представить.

Я ничего не сказал, я молчал после отправки *repo*.
Он сразу "всё знал" лучше, чем все остальные.

Что значит быть "разрушающим"

> *Ну, прямо сейчас я делаю YC и мой клиент — IBM.*

> *Да, и я знаю королеву Англии.*

> *Я когда-нибудь рассказывал вам о якудзе?*

> *Я ходил в лучшую инженерную школу и с тех пор я ничего нового не выучил.*

Конечно: позвольте мне процитировать девиз первой индонезийской хакерской группы, с которой я столкнулся в детстве: "We Can Do All What You Can't Do."
Я баг, которого ты не можешь убить.

И я здесь не ради *престижа*. И даже не ради *прибыли*.
Я здесь ради нашего **сообщества**.

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

ВЫ, вы были бы *луком*, этим ребёнком, который просто хотел показать себя и запустил PoC Exploit-DB из своей спальни как *новичок*. Чтобы доказать себя, выставить себя на обозрение, показать нам, как вы *в*. Я, я — это ребёнок, который впитал всё и вырос с этим.

Чёртовы *скиды*.
Чёртовы норми!

Вот кто мы: от Zone-H к уважаемой жизни.
От подземных туалетов интернета к жизни, где удаётся быть спокойно полезным.
Вы никогда не пережили бы такое.

От Ада до места, где мы сегодня.
Спасены исследованиями, красивыми ценностями, неблагодарной работой. Не того вида, который вас сияет. Того вида, переданного теми, кто умеет находить потенциал и спасать его перед тем, как будет слишком поздно.

НИКТО нас не спас, мы спасли СЕБЯ.
Спасибо тому, что говорит НАУКА.

Мы могли бы быть кровожадными, вместо этого мы стали одержимы навыками. Мы люди, которые посвятили свои жизни этим экранам, этой науке, пока вы издевались над подростками в чатах Dota или делали бог знает что, всегда показывая, всегда доказывая, что вы самый красивый, самый сильный, самый умный, самый властный.

Мы, мы придурки, но мы останемся онлайн до самого конца!

Мы — настоящая СЕМЬЯ!
И мы БЕЗ СКВА!

"Это полностью аутичный."
"Вы никогда ничего не будете стоить."
"Ты сумасшедший."
"То, что ты делаешь, полностью глупо."
"На самом деле, то, что ты пишешь, вообще не имеет смысла."

THE FUCK YOU THINK THIS IS?
You HOLLOW!
YOU HOLLOW, YOU UNDERSTAND ME?
YOU'RE WORTHLESS, YOU'RE FUCKING TRASH![2]

(btw: RIP, DG descendant...)

ТЫ БУДЕШЬ ЗАМЕНЁН И ВЕРНЁШЬСЯ К ПРЕСЛЕДОВАНИЮ СВОИХ СВЕРСТНИКОВ НА ДОТА И МАСТУРБИРОВАНИЮ НА МАКРОСЫ VBA КАК КУСОК ДЕРЬМА, КОТОРЫМ ТЫ И ЯВЛЯЕШЬСЯ!
ТЫ ГРЕБАНЫЙ НЕКОМПЕТЕНТНЫЙ УБЛЮДОК!
ТЫ ГРЕБАНЫЙ ПАРАЗИТ!
ТЫ ГРЕБАНЫЙ НЕСПОСОБНЫЙ!
YOU FUCKED WITH US!

Неподкупная

Более того, мой ответ будет сопровождаться цитатой от Дэвида Хейнемайера Хансона (DHH): "Fuck You."[3]

Да. Exactly: Fuck You. Это всё.
У МЕНЯ НЕТ ИНТЕРЕСА к программированию таким образом, и Я ВЛАДЕЮ Ocarina.

Обмен Ocarina означает **обмен машиной, которую я поддерживал полностью сам, для себя, поэтому с большой заботой**. Тем не менее, она распространяется как есть и останется такой же.
Это *моя* машина.

Я вложил всю свою ярость в неё, чтобы вылить всю мою любовь в неё.

Ocarina имеет направление.
Те, кто желает увести его в другую сторону со своим *wishful thinking*, вольны форкнуть его и никогда со мной не контактировать.

Внести вклад в проект с открытым исходным кодом, потому что это "круто" — это вполне зрелый менталитет, и терпимость к этому явлению наносит реальный ущерб.
Ни один подлинный участник не делает это ради "развлечения", а из *согласования личных интересов*.

Ocarina не является и никогда не будет *напыщенным* решением.
Ocarina является и **ОСТАНЕТСЯ решением для решения реальных проблем.**

Если вы не согласны, **вернитесь в r/unixporn[4], где вы принадлежите.** ✈️

Первое релевантное взаимодействие

Профессионал, который **действительно** посмотрел на структуру проекта, ответил: "*Это похоже на один из моих старых проектов Selenium, и мне это не нравилось.*" Это **именно** тот вид отзыва, который Ocarina построена адресовать.

Его первый инстинкт был приравнять POM к Selenium. Справедливо.
Но POM работает с любой технологией автоматизации.
Это "старомодно"? Абсолютно. И? Что дальше?

Пройдя через его разочарования, он обнаружил, как Ocarina их обрабатывает: "Подождите, это всё?"
Но на этот раз с кивком уважения. Он признал, что пришло время перестать пытаться быть *самым умным парнем в комнате*. После определённого момента, это убивает проект.

ИСТИННАЯ солидарность

Вместо того, чтобы идти *nerdy*, Ocarina фокусируется на решении *небольших проблем* без создания больших.
Вывод, сделанный из этого: "Ocarina практична."
Это цель Ocarina: её практичность.

Это как **мы** работаем и это действительно всё, что нужно, чтобы встать на борт с философией Ocarina.

Не нужно быть хакером, не нужно было столько страдать.
Это инструмент, который мы предлагаем, для страстных, действительно страстных людей.

Для тех, кто хочет строить, а не разрушать.
Для обоснованных, культурных людей, которые знают, кто они.

Для людей, похожих на нас, в конце концов.
На этот раз давайте объединимся.

Вдали от тех, кто разграбил то, чем мы были.

Почему не Playwright?

Его продолжение: "Почему не Playwright?"

Справедливый пункт.
Ocarina агностична, поэтому подключение Playwright легко.
Единственное ограничение: Ocarina никогда не будет поддерживать `async`/`await`.

Отступничество

В наше время люди запускают проекты как вы бы выскочили за пачкой сигарет.
Они пробуют, потому что это "модно". Итак, они все копируют друг друга, берут подписки на продукты друг друга просто чтобы взаимно надувать свои числа и опускать друг друга *Stripe* dispute rate.

Они думают, что они умны, но: реальность в том, что **каждый VC и каждый LP хорошо это знают,** и каждый играет свою роль в *fool's game*.

Но есть что-то, что они никогда не поймут. Они ничего "*underground*" не имеют: они просто **дети в кризисе идентичности.** Дети, которым мягко вручили *Powerpoint* в обмен на их мелки.

Урок для этих любителей: любой по-настоящему "*cutting-edge*" проект, какой бы он ни был, начинается с **памфлета**, уходит корнями в **идентичность**, и проходит через стадию подлинного **anti-marketing**. Но поскольку вы все *боитесь смущения вашей матери*, вы никогда не проходите через это.

Что касается меня, у меня нечего терять с точки зрения репутации, нет репутации, которую нужно создавать под этим именем. Только сообщение для доставки, как есть, без фильтра.

Пришло время пнуть всё ваше *Juicero Presses* в задницу.
С этой **знающей улыбкой**, улыбкой тех, **у кого наконец-то нашлось то, что нужно, чтобы сказать СТОП.**

IT является **тем, что вернул улыбку в нашу жизнь**, а не тем, что преследовало нас через pseudo-"patterns" всё более идиотские и недоступные.

Вы думали, что информатика умрет?
Далеко нет.

Анти No-Code

Код — это сырые данные. Проверяемы. Инспектируемы. **Белый ящик.**
Именно то, с чем AI знает, как работать с самых своих начал.

В наших собственных силосах мы уже попробовали это, даже до *IntelliCode* (2018). Еще в 2013 на нашей стороне: частный плагин *Emacs* (чёрт возьми), построенный одним из наших безумных учёных, R. Никто не мог постичь его уровень в *reverse engineering* и его лаконичность. Так же и в других областях... по правде говоря, он не был плох ни в чём и обладал необычайной способностью всегда писать *точно то, что нужно было написать*.

Конец мифа: у него был свой AI *autocomplete* и *auto-review*. 2013 год.

Он отверг большую часть сгенерированного кода, но сказал это: "*Я не возражаю против удаления 15,000 строк, если это спасает меня от написания 1,000 сами.*"

К 2013 году игра больше не была о том, чтобы быть бездумной "кодирующей" обезьяной.
Речь шла о культивировании себя, чтобы понять, как машина, вдохновленная нашим собственным рассуждением, помогла бы нам подняться выше этого тошнотворного, обреченного ООР.

Но "учёный" ООР не будет ни первым, ни последним трупом.
Информатика в основном развивается в одном направлении: кто свободно выбрал бы использовать Windows 95 сегодня?

И всё же, λ-calculus (1930) настолько мощен, что он вернулся в центр внимания сегодня, так же как AI (первая формализация искусственного нейрона в 1943, McCulloch & Pitts).
Необыкновенно в истории IT.

Последние достижения в AI перетасовали колоду для SaaS, который существует только чтобы скрыть сложность. Подлинно полезные продукты SaaS, а не бесконечные агрегаторы этого помпезного "цифровой трансформации", всегда находятся на правой стороне истории. Какими бы уродливыми они ни были.

Компании из экосистемы *Prisma* или *Vercel* тратят ~$200 в токенах на *vendor*, который они отправляют. Выходные за *vibe coding* — и они машут на прощание раздутым, переоцененным, вялым SaaS-платформам, в пользу *internal development* и возврата к **сырым данным**.

Декабрь 2025 года: Ли Робинсон, бывший Vercel, знакомое лицо из недавних презентаций *Next.js* и *Turborepo* и других, объявляет, что отказывается от *Sanity*, выбора CMS *Cursor* до этого момента. В пользу возврата к **сырым данным**, к простым файлам *Markdown*. Несколько токенов, некоторые хорошо выполненные *vibe coding*. Спасибо, до свидания.

Code is Law. Код *подменяет* собой Закон. Лессиг, 1999 год, как предостерегающий крик.
Затем подхвачено Ethereum в 2015 году, наоборот, как политический идеал. Значимый шаг вперёд в мире валюты, отчеканенной из факторизации цифр.

Анти-разработчики

2016 год: взлом The DAO, децентрализованного инвестиционного фонда на $ETH.
Почему?

Баги. Проклятые баги, вызванные проклятыми разработчиками.

Идеологический откат почти на 20 лет. Контрвласть сделала ленивый выбор "не понимать", тот же выбор, который позволил кучке некомпетентных проскользнуть под радаром.

И всё же это просто: в отличие от *Исследователей*, инженеры и их друзья из бизнес-школ полагаются на *пустые желания*.
Самые "престижные" школы научили их одному: выдавать пустые слова за "инженерию", в то время как те, кто действительно знает своё дело, называют это *программистской астрологией*.

Анти-нации-стартапы

И снова поседевшая дискуссия о "демократическом управлении кодом".
Но кто вмешивается?

Кадровики?
CEO?
Пресейл-менеджеры?

2022 год: ChatGPT. И всё же три года спустя основатели стартапов до сих пор продают *No-Code*, "усиленный AI". Чистый *cash burn* в погоне за *рыночной аномалией*, не более. *Альфа* инвесторов... иными словами: их *казино*.

Никого не называя, тем более что их результаты ожидаются во втором квартале 2026 года. Скажем лишь, что на основе *фундаментального анализа* эта ставка не имеет смысла. При этом нет нужды желать им зла. Это была бы не первая компания, которую мы видим с треском проваливающейся, и не последняя, которую мы увидим преуспевающей всем на удивление.

Факт остаётся фактом: код — самый суверенный актив, который у нас есть, и ценностное предложение, построенное на том, чтобы отобрать его у нас, заставляет нас морщить нос. Настоящая проблема — это разработчики, которым мы позволяем долбить по клавиатуре, не понимая, что они делают, как дрессированные обезьяны. Сегодня *Claude Code* превосходит 99% из них, и они воют. *Собака лает, караван идёт.*

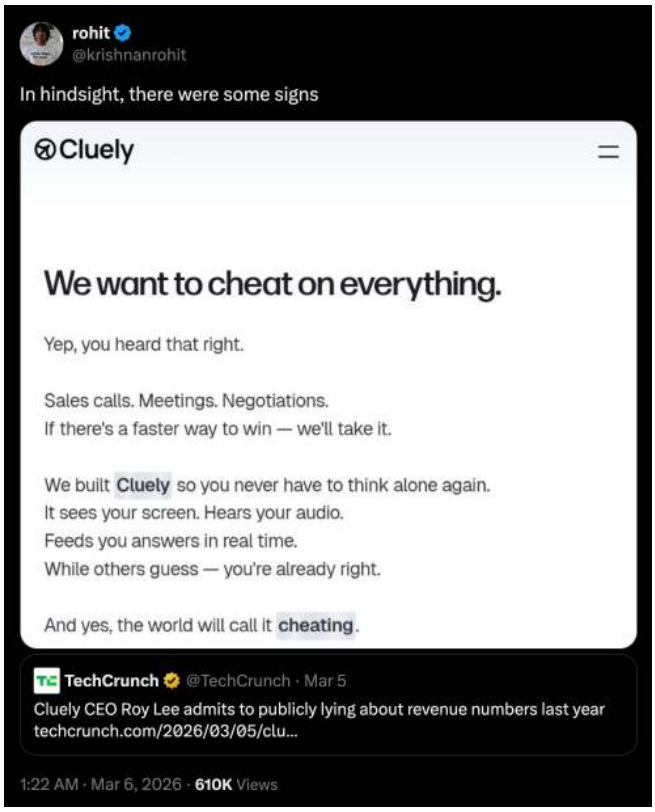
Истинная человеческая добавленная ценность никогда не была обозначена так чётко: *вкус* и *чувство ответственности*, качества, демонстративно отсутствующие у этих "профессионалов", которые топят нас в целых "идеологиях", прямо как те, кто *продаёт* то, чего не понимает.

Анти-хайп

В том же декабре 2025 года стартап, чьё имя останется непроизнесённым, был представлен в некоторых СМИ как "*AI No-Code тест, который унижает Selenium и заставляет BrowserStack трястись.*" Не меньше. Удачи им: предпринимательство — это мир ставок и случайности, где лучшие идеи заканчивают на кладбище так же часто, как и самые нелепые. Фундаментальный анализ там регулярно терпит поражение.

Ничто из этого, однако, не уведёт Ocarina от видения, которое формируется уже более 15 лет, вдали от высокомерия всех этих *детей хайпа*.

Что бы ни случилось, мы всё же желаем им судьбы, которая избавит их от унижения столь же стремительного и зрелищного, как то, что постигло основателя *Cluely*, и приглашаем их пересмотреть своё высокомерие.



CEO Cluely Рой Ли признаёт, что в прошлом году публично лгал о цифрах выручки. Затем Кришнан Рохит подчёркивает, что их ценностное предложение было мошенничеством с самого начала.

Сегодня вызов быть самым крупным мошенником становится всё менее и менее прибыльным. И мы этому рады. Снова, с AI, этим столь недостающим цифровым телом, прокричим это так же громко, как мы кричали "**HACK THE PLANET**" ещё в 99-м: **CODE IS LAW.**

Анти-slipologists

Слишком долго **программное обеспечение держали в заложниках** меньшинство, "1%", которые сочли уместным превратить его в **площадку для посвящённых**: "гиковские трюки", "ниндзя-техники", "объектная ориентированность". **Вердикт ясен: это не выдерживает или едва выдерживает.**

То, что занимает наши умы с 1930-х годов, **с изобретения λ-исчисления**, наконец-то масштабируемо до той степени, до которой мы всегда этого хотели.
Видела ли когда-нибудь наша индустрия столь элегантную формализацию, столь глубоко укоренённую в собственном наследии?

Недавние эволюции *системы типов* Python — центральны для того, что делает Ocarina возможной.
При этом не будучи и "будущим": **это устоявшаяся наука, но та, что пришла в нашу экосистему до жестокости поздно.**

Как и AI, который тихо делает способность "1%" причинять вред устаревшей.

Благодарность, причитающаяся этим технологическим достижениям, так же безмерна, как и ярость тех, кто восстаёт против них, кого в наших краях мы ласково зовём *slipologists*.

См. также: *Haters, Пола Грэма*.[5]

По всем этим причинам я принял решение **НИКОГДА не тратить своё время на споры, когда оно того не стоит.** Мне нравится спорить с людьми, **сосредоточенными на решении проблем и обладающими здравым суждением.**

Что до остальных: **делайте бессмысленные PR красными, а бессмысленные issue серыми, без сожалений.**
Некоторые люди просто не созданы мне нравиться, но так тому и быть.

Ocarina — *первый из моих проектов, который я открываю публике*, **но и другие тоже на подходе, включая куда более интересные.**

Из андеграунда,
Отныне и навсегда.

Peace out.[6]

"Идиот восхищается сложностью, гений восхищается простотой, физик пытается сделать это простым. Для идиота: чем сложнее что-то, тем больше он будет этим восхищаться; если ты сделаешь нечто настолько запутанное, что он не сможет это понять, он будет считать тебя богом, потому что ты сделал это таким сложным, что никто не может понять. Вот как пишут журналы в Академии: они пытаются сделать это настолько сложным, чтобы люди думали, что ты гений."

— Терри Дэвис, "самый умный программист из когда-либо живших"

"Чтобы обрести знание, прибавляй что-то каждый день. Чтобы обрести мудрость, убирай что-то каждый день."

— Лао-цзы

---

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

Глава 3

Первые шаги

Речь идет не о пункте назначения, а о путешествии.

Отказ от ответственности

> ***Примечание:*** *Эта книга предназначена, чтобы помочь вам познакомиться с предоставленным проектом* `ocarina-example`*, который остается* ***источником истины****, на который следует ссылаться во всех обстоятельствах.*

> ⚠️ *The Ocarina Holy Book НЕ является и никогда не будет "plug-and-play". Ocarina требует высокого уровня зрелости для использования. Поэтому мы сосредоточимся только на том, что действительно может быть трудным.*

Эта страница объясняет *путешествие*. После этого будет необходима практика.
📖 Получите канонический пример в качестве справки.[1]

1. Настройка проекта

Создайте новый проект Python, затем установите необходимые зависимости:

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

Затем создайте структуру папок.

2. Адаптеры

Ocarina построена вокруг системы адаптеров, которые пользователь отвечает за написание. Они позволяют фреймворку быть настроенным в соответствии с ограничениями и соглашениями каждого проекта.

Основные адаптеры, которые нужно создать:

- `act` *(требуется)*
- `test_campaign` *(требуется)*
- `test_suite` *(требуется)*
- `env_getters` *(опционально)*
- `match_page` *(опционально)*

2.1 EnvGetters

`EnvGetters` Ocarina централизует и типизирует доступ к переменным окружения. Он разделён на две категории:

- **Creds**: пары логин/пароль, выраженные как неизменяемые словари.

- **Values**: отдельные значения (строки).

```
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)
```

Как только этот адаптер будет на месте, получение значения или учетных данных выглядит так:

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

> ***Примечание:*** *Допустимые ключи предоставляются через два типа:* `EnvGetters[_CredsKeys, _ValuesKeys]`. *Если пользователь хочет использовать только* `.get_value()`, *достаточно типизировать* `_CredsKeys` *как* `Never`. *То же самое относится к* `_ValuesKeys`, *которые должны быть типизированы как* `Never`, *если пользователь хочет использовать только* `.get_credentials()`.

Наши акцессоры строго типизированы. Например:

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

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

2.2 Act

В Ocarina, `act` — это глагол, используемый для выражения каждого отдельного шага в тестовом сценарии. Его конструкция намеренно оставлена пользователю, по причинам, освещённым далее в этой книге (*hooks*).

Его минимальная форма выглядит следующим образом:

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

    return create_act(
        pom,
        action,
    )
```

2.3 TestCampaign

Адаптер `TestCampaign` намеренно минималистичен. Единственная информация, которую Ocarina не может вывести, — это **количество рабочих потоков**, то есть количество браузеров для параллельного запуска для данной кампании. Поскольку этот параметр также может быть передан непосредственно через CLI, необходимо только небольшое приложение:

```
@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,
        )
```

> *Тип* `WebDriver` *(Selenium или иначе) вводится здесь:* `OriginalTestCampaign[WebDriver]`.
> *И здесь:* `suites: Sequence[TestSuite[WebDriver]]`

> ✅ *Конечно, вставьте ВАШ адаптированный* `TestSuite` *здесь, а не встроенный в Ocarina.*

2.4 TestSuite

Это самый важный адаптер для понимания. `TestSuite` нативно выставляет большое количество параметров. Цель этого адаптера — создать **фасад** вокруг него: некоторые значения жестко закодированы раз и навсегда, другие опционально выставлены с разумными значениями по умолчанию. *Сужение*.

Аналогично:

```
@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,
        )
```

> *Тип* `WebDriver` *(Selenium или иначе) вводится здесь:* `OriginalTestSuite[WebDriver]`.
> *Также здесь:* `tests: Sequence[Test[WebDriver]]`
> *И здесь:* `drivers_pool: SeleniumWebDriversPool`

Временные ошибки

Концепция `transient_errors` имеет центральное значение для `TestSuite`.
Эти ошибки рассматриваются как **помехи**: если тест не прошел из-за исключения, указанного в `transient_errors`, он автоматически повторяется.
Максимальное количество попыток определяется `max_retries_per_test`.

Этот механизм делает выполнение тестов устойчивым к *flakiness*. Тесты, которые повторяются часто, четко видны в логах, позволяя разработчикам выявлять и исправлять источники нестабильности, вызванные неправильным использованием Selenium, условиями окружения вне области действия или другими внешними факторами.

Только ID и исключить ID

Эти два параметра позволяют условное выполнение тестов.
Они являются фильтрами на основе ID.

> ⚠️ ***Убедитесь, что включили их в этот адаптер, иначе эти флаги CLI не будут обработаны.***

2.5 MatchPage

`match_page` — это оператор Ocarina, разработанный для обработки страниц с недетерминированным рендерингом: баннеры cookie, антибот-вызовы, A/B тесты и т. д.

Его логика проста: **любое поднятое исключение интерпретируется как несовпадение и поэтому поглощается** `match_page`. Однако возможно исключить некоторые исключения из этого механизма, чтобы они распространялись нормально вверх по потоку выполнения.

Для согласованности `transient_errors` обычно должны попадать в эту категорию: они должны распространяться, а не молча подавляться.

Адаптер создается следующим образом:

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

3. Написание первого POM

Паттерн POM (*Page Object Model*) является хорошо установленным стандартом, который мы не будем переопределять здесь.

Вот как создать первый POM с 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
```

Несколько моментов стоит детализировать.

3.1 SeleniumTitleMixin

Любой объект, наследующий `POMBase`, должен реализовать метод `get_current_title`. `SeleniumTitleMixin` предоставляет эту реализацию прозрачно, без необходимости писать её вручную.

Его роль идет дальше: он также определяет атрибут `_driver` с типом `WebDriver` (Selenium), делая его **несовместимым с любым другим типом**. Попытка присвоить неправильное значение немедленно вызовет ошибку типа:

```
self._driver = "lol"

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

`SeleniumTitleMixin` также действует как **страж типа**. Аналогичные миксины могут быть созданы для других технологий браузерной автоматизации.

3.2 Возврат `self`

Каждый метод действия возвращает `self`. Это намеренный выбор конструкции в Ocarina, который следует последовательно соблюдать: он позволяет связывать методы в цепочки и плавно компоновать сценарии.

4. Написание соединителей

Соединители — это тонкий, но важный слой для читаемости сценариев. Они обертывают вызовы методов POM в явно названные функции:

```
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()
```

Они также могут быть составлены напрямую:

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

5. Написание первого тестового сценария

Все строительные блоки готовы.
Вот как собрать их в сценарий:

```
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)
    ),
)
```

Каждый шаг тестирования выражается через `act`, к которому привязаны обработчики `.failure()` и `.success()`.
Затем сценарий оборачивается в объект `Test` через `create_selenium_test`.

6. Создание тестовой суиты

Суита группирует набор тестов, которые должны быть выполнены против одного и того же пула драйверов:

```
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. Создание тестовой кампании

Кампания группирует несколько суит:

```
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. Создание тестового цикла

Цикл группирует несколько кампаний. Это единица выполнения наивысшего уровня:

```
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. Загрузка проекта

Вот полная точка входа для проекта:

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

Процесс выглядит следующим образом:

1. Аргументы, полученные из CLI, передаются в глобальное хранилище.
2. Создается пул драйверов: он управляет жизненным циклом веб-браузеров, работающих параллельно.
3. Определяется обратный вызов `_post_exec`: он запускается после тестов и плагинов, выводит результаты и выходит с кодом ошибки, если цикл не прошел.
4. Всё загружается внутри таймера, измеряющего общую продолжительность выполнения. Поток выполнения, таким образом: **цикл → плагины → post_exec**.

> ℹ️ *Плагины — это отложенные функции, переданные* `run_plugins`*.*
> `run_plugins` *принимает* `results` *в качестве аргумента,*
> *что сразу ясно из сигнатуры функции, что они запускаются как постобработка, как только результаты доступны.*

---

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

Глава 4

Первые сценарии

Язык — не реакционен и не прогрессивен; он попросту фашистский; потому что фашизм не препятствует речи, он заставляет говорить.

Конец недопустимых состояний

... Сделайте недопустимые состояния непредставимыми.

Мы рассмотрим, как `act`, `drive_page` и `match_page` работают при написании тестовых сценариев с Ocarina.

Act и drive_page

Канонический пример

Давайте начнём с примера, который мы постепенно сломаем:

```
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` выражает, что мы берём управление *одной* страницы.
Каждый *переход* становится явным через открытие нового `drive_page`.
Внутри, `act` выражает действие, выпущенное на этой странице: это *test step*. `drive_page` вариативен: он принимает столько вызовов `act`, сколько нужно, и запятая между каждым становится AND:

> *Открыть, затем проверить домашнюю страницу. И кликнуть CTA. Мы переходим на страницу: проверить, что мы на странице book-a-call.*

Иммунная система

Давайте попытаемся вызвать `verify_book_call_page` на `homepage`:

```
act(on_homepage, verify_book_call_page)

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

Действие несовместимо с его целью. Эта программа не *компилируется*.

Давайте забудем `.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
```

Давайте разместим `.success` сразу после `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'
```

Давайте поменяем местами `.success` и `.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'
```

Давайте объединим неоднородные вызовы `act` внутри одного `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
```

Многие команды спорят о *coding styles*.
Ocarina более ясна: если стиль не соблюдается, это не *lint* ошибка, это не предупреждение. Это не **компилируется**.

Ocarina применяет один и тот же шаблон для всех, и эти ошибки непосредственно появляются в редакторе через *mypy*: мгновенная обратная связь.

Приоритет тестовых сценариев — их однородность и простота. Это всё.

match_page

`match_page` обрабатывает ситуации, когда страница может быть отображена по-разному.

Давайте начнём с принципа *matchers*:

```
@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
```

***Matcher* минимально проверяет, верно ли что-то, как можно быстрее.**

> ⚠️ *Тем не менее, избегайте хвататься за сырой `.find_element(s)`: это прямая дорога к Selenium flakiness.*

Ограничение в 5 секунд не оказывает значительного влияния на горизонтально масштабируемую батарею тестирования: это не то, о чём стоит беспокоиться здесь. Также не

рекомендуется маскировать `verify` как *matcher*: **это два разных инструмента.**

Использование в сценарии:

```
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` находится на одном уровне с `drive_page` и компонуется так же. Его команда `then` ожидает цепь вызовов `drive_page` или `match_page`. Ветви определяются с помощью `when`.

`match_page` и `when` были добавлены поздно в Ocarina: Igoristan был настолько непредсказуем, что вариант использования стал очевидным.
Их интеграция была простой, доказательство гибкости грамматики: другие аналогичные структуры вполне могут следовать.

Повторения

Чтобы повторить цепь тестирования (например, чтобы протестировать несколько попыток несанкционированного доступа), просто умножьте список:

```
[
    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  # <- [!]
```

Фрагменты

Фрагмент — это функция `(driver, logger) -> TestChain`, которую можно внедрить до или после основной цепи, через `pre_test_scenarios_fragments` и `post_test_scenarios_fragments`.

Например, `login_without_otp_happy_path` — это фрагмент:

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

Внедрение в начале:

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

Внедрение в конце:

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

Оба параметра можно комбинировать, и каждый принимает список *фрагментов*, внедряемых в указанном порядке.

Псевдонимы

Сценарии могут стать тяжёлыми.
Поскольку всё декларативно, пользователь волен создавать псевдонимы:

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

Любое значение можно сделать псевдонимом и переиспользовать.
Это написание чистое: оно не производит немедленного *эффекта*.
Всё можно переобъявить в другом месте, переорганизовать в другом месте — лишь бы итоговая цепь соответствовала ожидаемому.

Глава 5

Первые дзюцу

И под льдом блеск реки...

Наборы данных

Управление тестом с набором данных просто с 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
]
```

Closure — это всё, что нужно.
Обратите внимание, что `Scenario` объявлена здесь изнутри. Это имеет смысл, так как вся суть в том, чтобы инкапсулировать её.

`multi_login_tests` — это *list* объектов `Test`, которые мы *unpack* в `TestSuite`, вот так:

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

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

Smoke тесты

Чтобы запустить smoke тесты в начале цикла с 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` принимает два значения (по умолчанию: `"fail-fast-on-first-smoke-campaigns-sequence-fail"`):

- `"fail-fast-on-first-smoke-campaigns-sequence-fail"`: как только одна кампания smoke тестов не пройдёт, остальные пропускаются.
- `"wait-for-all-smoke-tests"`: все кампании smoke тестов работают до конца, даже если один не прошёл на пути.

В обоих случаях основные тесты пропускаются, если какой-то smoke тест не прошёл.

Настройка и разборка

`Scenario` принимает два опциональных обратных вызова: `setup` и `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,
)
```

Жизненный цикл

1. `setup()`: работает перед `test_chain`. При отказе, `test_chain` пропускается и `teardown` всё ещё работает. Если каждая попытка не прошла из-за setup, тест помечается как **SKIPPED** (не **FAILED**),
2. `test_chain`: фактические шаги тестирования,
3. `teardown()`: всегда работает, даже при отказе. Ошибки логируются и игнорируются.

`setup` и `teardown` — это `Effect`.
Они предназначены для проблем инфраструктуры: заполнение базы данных, вызов API, очистка состояния...

Если нужна инкапсуляция: *closure*.

Proxy паттерн

Один вариант использования из канонического примера[1] это `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)
```

Идея: возвращайте *Web Elements*, которые ведут себя иначе для пользовательских взаимодействий. Нажатия клавиш, в этом случае.
Прозрачно для *type system*, прозрачно для *runtime*.

Что затем позволяет:

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

Или, с 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)  # <- [!]
```

Тот же принцип применяется к логгеру, маршрутизируя его к *sink*, например. Этот случай не канонически охватывается Ocarina.

Реактивное программирование: НЕТ

Тестовые сценарии Ocarina намеренно статичны.
Однако веб-приложение динамично, и иногда захват значения на лету для передачи на более поздний этап вполне легален.

Ocarina не отвечает на это. Ему это не нужно.

Архитектурный ответ

Что нам нужно здесь, это *in-memory cache*.

Мы генерируем ключи просто перед началом цепи тестирования и передаём их действиям POM. Действия записывают и читают через уникальный ключ.
Сценарий просто раздаёт их:

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

Вызовы API и блокировки

API и блокировки должны обрабатываться в POM.

> ⚠️ *Ocarina не поддерживает `async`/`await` и никогда не будет.*

Вызовы API: достаточно синхронного `requests`.
Блокировки: `threading.Lock`, если работает один процесс за раз, иначе достаточно распределённых блокировок Redis (`redis.StrictRedis` + `redis.lock`).

Профиль браузера

Некоторые случаи требуют передачи профиля через `--profile-path`:

- **Аутентификация прокси**,
- **Предзагруженные расширения**,
- **Локальные настройки** (язык, часовой пояс, сертификаты…),
- И т. д.

---

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

Глава 6

Первые реальные препятствия

Единственное разумное — это приспособиться к существующим условиям.

Случайные ошибки сервера

Случайные ошибки сервера — это крепкий орешек.
Во время особенно неприятного опыта работы мне пришлось иметь дело с окружением, которое регулярно отображало совершенно случайные страницы ошибок 500, независимо от того, какая часть приложения исследовалась.

В таких случаях ответ Ocarina живет прямо в создании глагола `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,  # <- [!]
    )
```

Хук `on_failure` был разработан именно для этого.
Всё, что нужно сделать, это создать несколько *предохранителей* и изменить ошибку, обёрнутую внутри `Fail`, чтобы запустить *повторное выполнение* любого теста, который не прошел из-за внешней причины.

Следующий шаг должен быть знаком:

```
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,
        )
```

Наконец, если `match_page` также используется в проекте и общая переменная `transient_errors` нежелательна, не забудьте добавить эти новые определения ошибок в параметр `raised_exceptions` конструктора `match_page`.

Журналы автоматизированного тестирования предоставят хорошую отправную точку для поднятия этих проблем с командой.

Случайные ошибки в пределах шага

Думая, что я оставил эту чушь позади, я двинулся дальше, только чтобы найти нестабильные формы и системы аутентификации, которые работали только половину времени.

Здесь ответ Ocarina другой: мы делегируем ответственность на 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
```

Это также поднимает вопрос о *connectors*: как передать параметры к ним?

```
"""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
```

Просто верните `def` с ожидаемой сигнатурой, внутри функции, которая захватывает параметры.
Это *closure*[1].

Случайные ошибки Selenium

Selenium даёт массу возможностей выстрелить себе в ногу: *race conditions*, ошибки *stale element* и так далее.

Ответ здесь **прагматичен**: добавьте `WebDriverException` непосредственно в `transient_errors`, с щедрым количеством повторов (8, что означает 9 жизней, как кошка 🐱).

Захватите все ошибки Selenium и смотрите повторы в логах.
Отсюда становится легко выявить тесты, которые могли бы использовать какое-то улучшение.

Дискретные случайные ошибки

Еще более удивительно: приложения, выводящие тосты об ошибках без видимой причины, или формы, сообщающие об ошибках валидации на идеально правильных входах, без фактического блокирования потока.

Эти ошибки самые трудные для поимки именно потому, что они *безболезненны*. Вы не можете просто заметить сбой и добавить *политику повтора*, ожидая, пока ошибка будет исправлена. Они, по сути, невидимы.

Что остаётся? Портить тестовые сценарии или хвататься за «ниндзя-техники».
Ocarina отказывает обоим.

Используйте *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` — это *callback*, который *watcher* будет вызывать каждые 0.8 секунды (`poll_interval`).

Давайте уточним несколько вещей.

Использование JS

Watcher терпим к ошибкам: он молча поглощает исключения.
Поэтому нет никакой пользы от использования функции Selenium для захвата элемента страницы, это только добавило бы ненужный багаж.
Использование собственных функций Selenium означало бы работу с проблемами *implicit timeout*.

Переход прямо через *Javascript* обходит всю внутреннюю логику *polling* и держит исполнение *watcher* как неблокирующее возможное для теста, работающего на том же *driver*.

Весь трюк становится невидимым, так как занимает всего несколько миллисекунд.

Отпечатки пальцев

Watchers выставляют простой кэш строк, разработанный специально для этой потребности: если одна и та же ошибка остаётся видимой и обнаруживается каждые 0.8 секунды, нет смысла видеть её повторно в скриншотах, отчётах и логах. Отпечаток пальца позволяет вам игнорировать то, что вы уже видели.

Отчет

В конце *callback* он вызывает: `watcher.report`.
Этот вызов управляет:

1. Логированием трения, обнаруженного *watcher*,
2. Созданием скриншота как следа того, что было обнаружено.

HumanizedDriver

Ничто не мешает нам привязывать поведение к *logger* или *driver*. Здесь, поскольку форма капризна, мы выбираем медленный, "очеловеченный" тест: набор текста с опечатками, исправлениями, колебаниями. Мы просто оборачиваем *driver* в *proxy*, `HumanizedDriver`.

Гейзенбаги конкурентности

Мои поиски на этом ещё не закончились.
Однажды я наблюдал, как коллеги считали до трёх, прежде чем все одновременно кликнуть, чтобы запустить одно и то же действие, прямо там, в офисе. Я поймал себя на размышлениях о смысле своей жизни. И всё же, делая это, они действительно умудрялись воспроизводить баги.

Это поведение можно воспроизвести с помощью Ocarina.
По умолчанию Ocarina агрессивна.

Её опция `saturate_workers` принудительно случайно клонирует тесты внутри набора.
Всякий раз, когда в *DriversPool* доступно больше *workers*, чем тестов для запуска в наборе, Ocarina будет случайным образом клонировать тесты, запускать все драйверы и назначать каждому из них тест для выполнения.

Эту опцию можно включить из функции `bootstrap`. Её также можно переключать индивидуально, либо на уровне набора, либо на уровне кампании.
При конфликте последнее слово за самым глубоким элементом в иерархии.
Например, если кампания отключает опцию, а набор её включает, приоритет за набором.

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

Также возможно временно создать набор с одним-единственным тестом, чтобы максимизировать точность нацеливания.
Или запустить цикл на нескольких машинах одновременно (горизонтальное масштабирование).

Помимо вопросов конкурентности, этот механизм клонирования также нацелен на то, чтобы гарантировать, что проходящие тесты ничем не обязаны случайности. Этот эффект усиливается степенью горизонтального масштабирования и количеством задействованных workers.

---

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

Глава 7

Расширяемость

Если мы не верим в свободу выражения мнений для людей, которых мы презираем, мы вообще в неё не верим.

ValidationChain

Используется в POMs, `validate` позволяет выражать инварианты как цепи. Выполнение **отложено**: `.execute()` должен быть вызван явно.

Результат выставляет `is_valid`, `errors` и `validated_values`. Это инертно по умолчанию. `.raise_if_invalid()` выбрасывает исключение, если необходимо.

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

Объединение инвариантов

Несколько утверждений на одном значении:

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

Объединение валидаций

Несколько валидаций на разных значениях:

```
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()
```

Переиспользуемые инварианты

Чтобы учесть повторяющуюся валидацию, создайте *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
    )
```

```
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()
```

Соглашение: `FrameworkInvariantValidator.create` для технических инвариантов, `BusinessInvariantValidator.create` для бизнес-инвариантов.

Пользовательские утверждения

Без аргумента:

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

С аргументом:

```
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
```

Тип безопасности

Проверка типов ловит утверждения, несовместимые с типом значения:

```
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 и Failure

`.success` и `.failure` каждый принимают *effect* для выполнения.
Канонический пример[1] реализует несколько обработчиков: простой логинг ошибок, логинг ошибок с текущим URL, логинг успеха и логинг успеха со скриншотом (+ URL).

```
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
```

Другие обработчики стоит рассмотреть:

- `create_log_error_with_retry_hint`: сигнализирует о *transient error* и, следовательно, о возможности flakiness,
- `create_log_error_and_send_alert`: отправляет webhook при неудаче, не загрязняя сам тест,
- `create_log_success_and_record_timing`: захватывает временную метку завершения для измерения фактической длительности шага (комбинируется с `on_run_effect` из `create_act`),
- И т. д.

Также стоит рассмотреть *combinator*.

Плагины

`bootstrap` позволяет плагинам после выполнения запускаться на основе результатов цикла тестирования. Например, `generate_docx_proof` проходит по дереву логов и генерирует один документ Word (тестовое доказательство) для каждого тестового случая, встраивая скриншоты и преобразуя временные метки UTC в локальное время.

Идея: плагины переставляют артефакты, произведённые на ходу, в другую форму. Плагин, генерирующий отчет веб-панели, будет естественным выбором, например.

Расширяемая грамматика

Грамматика тестовых сценариев построена на одном типе: `ChainRunner[T]`. Сценарий — это `list[ChainRunner]`, выполняемый последовательно, с коротким замыканием на первом отказе. `drive_page` — это просто тонкая обёртка вокруг `chain_actions`, которая строит `ChainRunner`. Любая функция, возвращающая `ChainRunner`, подключается без касания фреймворка.

`match_page` был добавлен, чтобы обрабатывать страницы с переменным состоянием (опциональные баннеры, A/B тесты, страницы обслуживания...): он оценивает условия по порядку и запускает первую совпадающую ветвь.

Другой пример был бы `skip_if`: намеренный пропуск части сценария на условии без отказа (вернёт нейтральный `Ok`), полезный для опциональных шагов, зависящих от окружения или данных.

Единственный контракт расширения: верните `ChainRunner`.

---

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

Глава 8

Использование Ocarina с AI

Привет, это я, твой новый лучший друг!

Рабочая конфигурация: полный цикл тестирования, построенный рядом с Claude Code и Ocarina, против общедоступной демонстрации Katalon CURA.

📖 Получите пример AI в качестве справки.[1]

Три духовных камня

1. `CLAUDE.md` в корне проекта.
2. `skills/` с одним `<name>/SKILL.md` для каждой процедуры.
3. Правило проверки: каждое утверждение SUT исходит из наблюдения (probe, `gh api`, `curl -v`), никогда из вывода.

`CLAUDE.md`

Два варианта. `CLAUDE.md` полный (правила + макет проекта, иерархия, соглашения, форма CI, шаблон PR). `CLAUDE.slim.md` только правила. Slim когда контекст тяжёлый; полный для onboarding и обзоров. Полный выигрывает при разногласии.

Шаги Onboarding (venv, `pip install`, `ruff` / `mypy` / `pre-commit`, runner smoke-check) находятся в `setup-environment`.

Правила:

Тестирование безопасности функционально и статично, никогда не активно. Нет полезных нагрузок, нет разработанных запросов, нет манипуляции DOM DevTools. Сценарии чёрных шляп проходят через обычный пользовательский интерфейс.

Используйте константы. Именованные значения не встроены.

Наборы данных — это решения людей. Предложение не работает.

Проверьте поведение SUT эмпирически. Probe, `gh api`, или `curl -v`. Никогда не вывод. Переделайте каждый раз: probe отвечает только за то, что он работал; предыдущий диагноз только для этого запуска.

Каждое правило несёт однострочный "*почему*".

`skills/`

Один файл Markdown для каждого навыка, YAML frontmatter + тело. Девять семейств.

Обзор (13)

Статические чтения; поверхностные находки.

- `review-spec-gaps` — вопросы уточнения на FRD.

- `review-watcher-misuse` — `watcher.report(...)` против соглашения только-отрицательное.
- `review-compartmentalisation-leaks` — URLs, selectors, волшебные числа не на месте.
- `review-dead-code` — неиспользуемые connectors / POMs / сценарии / suites / фрагменты / константы; для каждой находки: удалить, инкубировать (`<source-root>/incubator/`, дерево зависимости сохранено), или хранить.
- `review-report` — классифицировать каждый FAIL / SKIP для одного запуска.
- Плюс: `review-type-ignore`, `review-match-candidates`, `review-unverified-transitions`, `review-submit-dispatchers`, `review-comment-drift`, `review-suite-stability`, `review-intent-collisions`, `review-watcher-emissions`.

Анализ (4)

- `analyse-flakiness` — расширить сеть transient-error; хронические смерти — это реальные flakes.
- `analyse-fixture-flakiness` — инструмент setup/teardown; поверхностная перекрёстная контаминация тестов.
- `analyse-watcher-flakiness` — с/без каждого watcher, sweep интервала.
- `analyse-screenshot-flakiness` — группировка по `(test, step, browser)`, заметить различия.

Чёрная шляпа (6)

- `business-attack-ideation` — сбрось продукт.
- `incoherence-attack-ideation` — каждый шаг легален, набор невозможен.
- `persistence-attack-ideation` — повторные повторы на заблокированных действиях.
- `permission-appropriateness-audit` — уместна ли сама модель доступа?
- `bfcache-exposure-ideation` — BFCache атаки.
- `lateral-resource-ideation` — IDOR через адресную строку только.

Понимание (4)

- `assess-test-base` — каталог набора.
- `assess-ecosystem` — ограниченные общественные исследования, бюджет токенов закрыт.
- `understand-sut-constraints` — SUT границы, которые разрушают параллельные тесты.
- `understand-ocarina` — пройдите документы.

Выбор (3)

По mtime, никогда имени файла.

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

Автор (8)

Каждый производит готовый результат.

- `empiricism` — проверьте перед кодированием; не перезаписывайте intentional-fail gap тесты.
- `write-a-probe` — одноразовый скрипт, gitignored.
- `write-test-strategy` — генерируйте документ test-strategy из набора (scope, types, таблицы покрытия, дерево цикла, pass/fail, gaps, CI матрица).
- `extend-coverage` — расширьте покрытие из существующих активов.
- `update-frd-and-tests` — распространите обновление spec.
- `manual-reproduction-guide` — человеко-запускаемый repro.
- `manage-backlog` — `BACKLOG.md`.
- `pr-report` — PR-type-aware отчет.

Рефакторинг (2)

- `refactor-fragmentation` — DRY по предпочтению пользователя.
- `introduce-pom-retries` — POM-internal повторы с разбивкой на два теста (first-try + with-retries).

Состояние (1)

- `question-state` — опросить окружение, прежде чем доверять результату.

Настройка (1)

- `setup-environment` — venv, dev tooling, набор скиллов Ocarina, скопированный в директорию скиллов Claude Code, пути драйвера в `CLAUDE.local.md`, pre-commit loop, runner smoke-check.

Запуск (1)

- `propose-visual-review` — перед локальным запуском предлагает `--not-headless` (наблюдать, как браузер отыгрывает сценарий) против headless (в форме CI). Составляет команду; пользователь запускает.

Повторяющиеся цепи

Suite не зелёная: `review-report` → `analyse-*` → `write-a-probe` → находка попадает в `IDENTIFIED_GAPS.md` / FRD / комментарий сценария → probe удалён.

Сценарий чёрной шляпы выглядит многообещающе: `empiricism` → `extend-coverage` (часто intentional-fail).

Изменения Spec: `update-frd-and-tests` (FRD первый, тесты следуют). Gap тесты переделаны, не перевёрнуты.

Нужен новый примитив Ocarina: сначала `understand-ocarina`, затем написание.

Дисциплина

Показывайте, не применяйте. Навыки производят; пользователь решает.

Эмпирический, не утверждающий. Каждое утверждение SUT наблюдается, процитировано, датировано. Ритуальная фраза: *"Fair point, I'm assuming. Let me verify empirically."*

Gap-тесты переделаны, не перекрашены в зелёный. Инвертируйте утверждение, переименуйте, переместите строку strategy-doc, логируйте разрешение в `IDENTIFIED_GAPS.md`. Одно движение через `update-frd-and-tests`.

Watcher emissions только отрицательные сигналы. Watcher, выпускающий *"login succeeded"* разрывает контракт.

Распределённо когда нехватка распределённая. Если рабочие состязаются на SUT-ограниченном ресурсе (сессии, слоты, квоты), координируйте через распределённые примитивы. Иначе worker-local in-memory cache в порядке — при условии, что ключи не могут столкнуться и генерирование потокобезопасно.

Mtime, не имя файла. UUID суффиксы случайны; `pick-*` сортирует по mtime.

Чем эта настройка не является

- Не генерирует тесты автономно.
- Не латает галлюцинации в CI; отказ запускает `review-report` + `analyse-*`.
- Не переписывает spec; только `update-frd-and-tests` делает, с линией пересмотра.
- Не запускает активные тесты безопасности. Никогда.

Выставленные ресурсы

- 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