

REFACTOR LIKE A SUPERHERO



**HOW TO REFACTOR APPS
EFFICIENTLY AND WITHOUT PAIN**

BY ALEX BESPOYASOV

Предисловие

Эта книжка выросла из моего доклада «Рефакторинг на максималках»⁹, который я готовил в январе 2022 года. Для этого доклада я собрал различные эвристики и техники рефакторинга, которыми хотел поделиться.

При подготовке стало очевидно, что в 40-минутный слот не получится затолкать всё, о чём хотелось бы рассказать. Материала оказалось много — пришлось его ужимать, фильтровать и обрезать, чтобы уместиться в отведенное время и показать полезные практики.

Отфильтрованный материал было жалко. В нём оставались детали и уточнения, когда что-то применимо, а когда не очень. Я решил, что будет полезнее не выбрасывать всё, что не вместилось, а изменить формат и выпустить в виде небольшого сборника. Так появилась эта книжка.

На кого это рассчитано

В первую очередь этот текст может оказаться полезным разработчикам веб-сервисов, пользовательских приложений и интерфейсов, которые пишут на высокоуровневых языках и имеют пару-тройку лет опыта.

Разработчики библиотек тоже могут найти для себя полезные подходы, но я буду фокусироваться на пользовательских приложениях и веб-сервисах — в этой области у меня больше опыта.

Я не учитываю специфику низкоуровневой разработки. Часть эвристик могут противоречить хорошим практикам или даже вредить низкоуровневому коду. Если вы пишете близко к железу, будьте аккуратны, читайте на свой страх и риск 😊

Чем этот текст не является

Я не претендую на единственно правильный способ рефакторить или писать код. Если у вас много опыта, вероятно, о большей части описанных техник вы уже знаете и у вас есть своё мнение на их

счёт.

Также я не стараюсь написать «мануал», который будет универсально применим во всех проектах. Моё восприятие, привычки и метод работы искажены моим опытом разработки. Ваш опыт, проекты и привычки могут сильно отличаться от моих, поэтому взгляды тоже могут не совпадать. Это нормально.

Цель этой книжки в том, чтобы описать набор практик, эвристик и подходов, которые в своё время помогли *мне* начать писать код, который *кажется* хорошим мне и командам в проектах, где я участвовал.

Не все практики надо применять всегда. Решение, применять идею или нет, сильно зависит от проекта, задачи, ресурсов и цели рефакторинга. Страйтесь выбирать те идеи, которые принесут больше выгоды при меньших затратах.

Если что-то из книги кажется вам полезным, обсудите это с коллегами и другими разработчиками. Убедитесь, что у вас с командой одинаковое мнение о пользе и издержках выбранной идеи. Не применяйте то, что вызывает в команде споры или разногласия.

Ограничения и применимость

Все описанные техники и приёмы — это большая компиляция прочитанных книг и моего опыта в разработке. Большую часть времени я посвятил разработке пользовательских приложений среднего и иногда большого размера.

Мой опыт накладывает отпечаток на то, как я вижу хороший код. По сути вся книжка — это такой снимок моего восприятия разработки в 2022 году. Если вы читаете это в будущем (привет!), возможно, моё мнение поменялось, так бывает.

К слову

Я буду обновлять текст книги по мере изменения мнения, но не могу гарантировать, что буду это делать оперативно и без задержек.

При чтении книги помните о когнитивных искажениях автора. Мысленно сравнивайте техники со своим проектом и думайте об их применимости.

Что хорошо знать перед прочтением

В тексте я подразумеваю, что вы уже знакомы с самим понятием рефакторинга [🔗] и что у вас есть пара лет опыта программирования. Я ожидаю, что вы уже сталкивались с проблемами декомпозиции задач, «протекающих» абстракций, разграничением ответственности между модулями и т.д.

Я рассчитываю, что вы слышали о какой-то части «программистских словечек» из вот этого списка:

- Разделение ответственности
- Зацепление и связность
- Декларативность
- Слои абстракции
- Разделение на команды и запросы
- Ссылочная прозрачность
- Функциональный пайплайн
- Функциональное ядро в императивной оболочке
- Слои архитектуры, порты и адаптеры
- Направление зависимостей
- Неизменяемость и отсутствие состояния
- 12-факторные приложения

Вам не обязательно знать их все, потому что мы будем раскрывать смысл техник по ходу книги. Но будет хорошо, если вы имеете представление, в чём их польза и основной смысл.

Зачем нужна ещё одна книга

Книг о рефакторинге много, зачем нужна ещё одна?

По большому счёту — ни зачем. Все принципы, техники и причины их появления описаны в других книгах и, скорее всего, гораздо более подробно, детально, логично и корректно.

Мне этот текст нужен в первую очередь самому:

- для систематизации того, что я знаю сам;
- для возможности сослаться на конкретное место при дискуссии о какой-то проблеме;
- для фиксации уровня знаний в конкретном моменте времени, чтобы понимать куда расти.

Но я подумал, что возможно, это может оказаться полезным кому-то ещё. Так этот проект и появился.

1. Доклад «Рефакторинг на максималках», <https://bespoyasov.ru/talks/refactor-like-a-superhero/>

2. Рефакторинг, Википедия, <https://ru.wikipedia.org/wiki/Рефакторинг>

Введение

Рефакторинг требует ресурсов. Количество этих ресурсов зависит от размера проекта и качества кода в нём. Чем больше проект и хуже код, тем сложнее наводить в нём порядок и больше ресурсов может для этого потребоваться.

Чтобы обосновать вложение ресурсов и найти баланс между затратами и выгодой, нам надо понять пользу и ограничения рефакторинга.

Польза для разработчиков

Порядок в кодовой базе — это инвестиция в свободное время разработчиков в будущем. Чем проще и понятнее код, тем меньше времени будет уходить на исправление багов и новые фичи.

Разработчикам могут быть важны разные свойства кода. Например, нам может быть важно:

- Быстрее находить куски кода, отвечающие за конкретные части приложения.
- Исключить разнотечения о работе кода, недопонимание и конфликты в команде.
- Легче проводить код-ревью и сверять код на соответствие бизнес-требованиям.
- Добавлять, изменять и удалять код без регрессий и лишних усилий.
- Уменьшить время на поиск и исправление багов, сделать процесс отладки удобнее.
- Упростить исследование проекта для новых разработчиков.

Это неполный список. Конкретной команде могут быть важны и другие свойства, они могут варьироваться от проекта к проекту.

Регулярный рефакторинг помогает уделять внимание характеристикам кода заранее, до появления проблем с ними. Это делает ежедневную работу удобнее и предотвращает «большие рефакторинги» в будущем. Такой процесс даёт разработчикам больше свободного времени и ресурсов.

Польза для бизнеса

В идеально организованной разработке необходимости «продавать» рефакторинг бизнесу нет. В таких проектах регулярное улучшение кода «вшито» в процесс разработки и плохой код не накапливается. Отдельно «объяснить пользу бизнесу» в этом случае не нужно.

Но есть проекты, где разработка по разным причинам организована иначе. В таких проектах, как правило, копится легаси.

Мы можем чувствовать необходимость улучшить код, но у нас может не хватать на это ресурсов. Предложение «взять недельку на рефакторинг» вызовет конфликт интересов, потому что для бизнеса оно звучит, будто «целую неделю не будет происходить ничего полезного». В этих случаях нам и может понадобиться «продать» идею улучшения кода.

Польза рефакторинга для бизнеса неочевидна, потому что она не мгновенна. Польза проявляется «через какое-то время», какое именно — предсказать трудно.

Обычно, чтобы продать идею рефакторинга бизнесу, я стараюсь говорить на языке бизнеса и продавать *не процесс, а результат*. Что именно мы получим в результате потраченного времени:

- Сможем быстрее находить и исправлять ошибки, это уменьшит количество разочарованных пользователей.
- Начнём реализовывать новые фичи раньше конкурентов, это будет генерировать новых пользователей и прибыль.
- Будем лучше понимать требования, это позволит раньше реагировать на непредвиденные проблемы.
- Избавимся от текучки кадров, потому что от приятного кода разработчики не бегут.
- Сделаем онбординг быстрее и понятнее для новых разработчиков, это позволит им приносить пользу раньше.

Мы можем использовать различные метрики для измерения качества кода. Опираясь на эти цифры будет проще обусловить необходимость рефакторинга. А снижение затрат при регулярном рефакторинге поможет плавно внедрить его в процесс разработки.

«Плохой» и «хороший» код

Мне сложно назвать список *объективных* характеристик хорошего кода. Таких характеристик мало, и у них есть ограничения в трактовке и применимости.

Среди таких характеристик можно выделить цикломатическую сложность и количество зависимостей. Но мы поговорим о них отдельно в будущих главах.

Это не новая проблема. В большей части книг, что я прочёл, хороший код описывают субъективно:

- У Физерса хороший код «читаемый, поддерживаемый и приятный»; ☺
- У Фаучера — «читаемый, лаконичный и простой»; ☺
- У Мартина — «элегантный, простой и читаемый». ☺

Разные авторы используют разные слова, но можно заметить, что все делают упор на «читаемость».

Есть исследования, которые пытались определить, что такое эта «читаемость». ☺ ☺ ☺ Однако их проблема в маленькой илиискажённой выборке, поэтому делать выводы об универсальных правилах «хорошего» кода сложно.

На практике я стараюсь искать не «хороший», а «плохой» код. Это проще, потому что в его поиске мне помогают эвристики и «когнитивные костили».

Когнитивными костилями я называю ощущения, которые появляются при чтении плохого кода. Я считаю, что коду нужен рефакторинг, если при чтении возникает одна из этих мыслей:

Тяжело читать

- Нам тяжело читать код, если в нём беспорядочное форматирование, он «грозный», запутанный, шумный.
- В коде много лишних деталей, нет явной точки входа.
- Сложно проследить последовательность выполнения, нужно прыгать между экранами, файлами, строками.
- Код непоследовательный, не отвечает правилам, принятым в проекте.

Тяжело менять

- Код тяжело менять, если при добавлении новой фичи нужно изменить много файлов или перепроверить всё приложение.
- Нет уверенности, что можно беспроблемно удалить конкретный кусок кода.
- Нет явной точки входа, нельзя соотнести фичу приложения и конкретный модуль.
- Слишком много бойлерплейта или копипасты.

Тяжело тестировать

- Код тяжело тестировать, если для тестов нужна «навороченная инфраструктура» или нужно мокать много функциональности.
- Приходится имитировать работу всей программы, чтобы проверить одну функцию.
- Для теста нужны тестовые данные, которые не относятся к задаче.

«Не помещается в голову»

- Код не помещается в голову, если сложно уследить за всем, что в нём происходит.
- К середине модуля сложно вспомнить, что было в начале.
- При чтении «кипят» голова, схемы работы на бумажке не помогают.

Запахи кода

Часть описанных проблем умные люди уже оформили в виде запахов кода. *Запахи* — это антипаттерны, которые приводят к проблемам. ☣

Против запахов уже разработаны решения. Иногда нам достаточно посмотреть на код, найти в нём запах и применить конкретное решение против него.

О ЗАПАХАХ

Чаще всего примеры запахов приводят в коде, написанном в ООП-стиле, что может быть не так полезно в JavaScript-мире. Тем не менее часть запахов достаточно универсальна и применима не только к ООП, но и к мультипарадигменному коду.

-
1. "Working Effectively with Legacy Code" by Michael C. Feathers,
https://www.goodreads.com/book/show/44919_Working_Effectively_with_Legacy_Code
 2. "The Art of Readable Code" by Dustin Boswell, Trevor Foucher,
<https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
 3. "Clean Code" by Robert C. Martin, <https://www.goodreads.com/book/show/3735293-clean-code>
 4. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies,
<https://github.com/reydne/code-comprehension-review/blob/master/list-papers/AllPhasesMergedPapers-Part1.md>
 5. Code Readability Testing, an Empirical Study,
https://www.researchgate.net/publication/299412540_Code_Readability_Testing_an_Empirical_Study
 6. How Readable Code Is, a Readability Experiment <https://howreadable.com>
 7. Code Smells, Refactoring Guru, <https://refactoring.guru/refactoring/smells>

3

Прежде, чем начать

Чтобы рефакторинг прошёл быстро, без регрессий и не нагружая команду большим количеством работы, перед его началом код стоит *исследовать и подготовить* к изменениям. А именно:

- Определить границы рефакторинга.
- Покрыть выбранную часть кода тестами.
- Настроить линтеры и компилятор.

В этой главе обсудим, как упростить подготовку кода и зачем нужен каждый из перечисленных пунктов.

Определить границы

Под границами рефакторинга мы будем понимать места стыка между кодом, который мы будем менять, и всем остальным. Они определяют, в каком месте наши изменения должны остановиться, то есть какой код мы менять *не будем*.

Такое ограничение важно по двум причинам:

- Мы хотим оставаться в рамках *временного и ресурсного бюджета*, который у нас есть.
- За маленькими изменениями проще следить и понимать, что именно сломало работу приложения.

Наметить границы в коде бывает сложно, особенно, если модули беспорядочно переплетены друг с другом. Мне в таких случаях помогает обратить внимание на *данные и зависимости*, с которыми работает код.

Чем сильнее отличаются данные, с которыми работают куски кода, тем выше вероятность, что это разные «единицы смысла» — самостоятельные части программы. Место стыка этих частей и будет границей, которая ограничит распространение изменений.

Физерс в «Эффективной работе с легаси» называет такие места «швами». Я иногда буду использовать этот термин как синоним. ☺

Границы не дадут рефакторингу модуля превратиться в «долгострой» и помогут интегрировать изменения в основную ветку репозитория чаще.

ПОДРОБНЕЕ

Чуть подробнее о поиске и использовании границ мы поговорим в следующих главах.

Покрыть тестами

Код внутри выделенных границ нужно покрыть тестами. С их помощью мы будем проверять, что ничего не сломали во время рефакторинга. Чтобы тесты приносили больше пользы, я стараюсь выполнить несколько условий:

Выявить больше крайних случаев

Крайние случаи помогут избежать регрессий и убедиться, что мы не сломали работу кода в «экзотических» обстоятельствах. («Экзотические» баги чинить сложнее. ☺)

Чем крайние случаи разнообразнее, тем легче подобрать и систематизировать тестовые данные для различных ситуаций. Знания о поведении приложения в этих ситуациях пригодятся нам и в будущем, после рефакторинга.

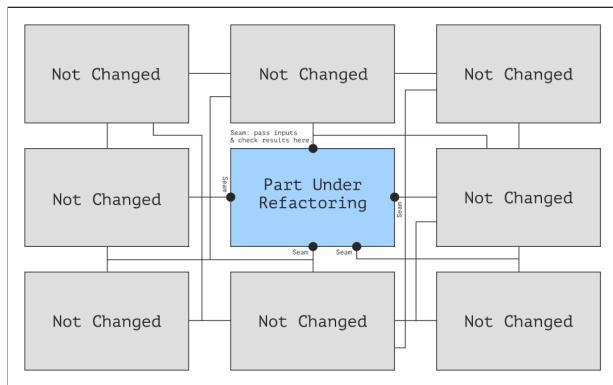
Определить явные и неявные входные данные

Явные входные данные — это аргументы функций или методов. Неявные — зависимости, общее или глобальное состояние, контекст работы функций и методов. Систематизация входных данных упростит составление тест-кейсов.

Конкретизировать желаемый результат

Во время рефакторинга мы не меняем функциональность, поэтому желаемый результат — это фактическое поведение программы. Зафиксировать это поведение мы можем в виде данных (например, результата работы функции) или желаемого побочного эффекта (изменения состояния или вызова API).

В идеале результат должен находиться *на границе* части кода, которую мы рефакторим. Проверять такой результат проще, а количество затронутого кода будет минимально.



Если результатом находится на границе, его проще проверять

Настроить автотесты

Нам потребуется тестировать каждое изменение. Тестирование вручную будет утомлять, из-за чего мы можем начать лениться или забывать о проверке изменений.

У автоматических тестов таких проблем нет. Мы можем настроить перезапуск тестов на каждое сохранение кода и запустить их перед началом рефакторинга. Так результат проверки всегда будет перед глазами, и мы раньше заметим, какое изменение сломало работу кода.

Как выбрать вид тестов, зависит от ситуации и не так важно, как их наличие. Если можно обойтись юнит-тестами, то я предпочту использовать их. Если приходится тестировать работу нескольких модулей, может понадобиться интеграционный или E2E тест.

Основной смысл — именно в автоматизации. Чем меньше проверок мы будем делать руками, тем меньше будет вероятность человеческой ошибки.

Ужесточить настройки линтеров и компилятора

Этот пункт опциональный, но очень нравится мне лично.

Более агрессивные настройки линтера или компилятора помогают подмечать случайные ошибки и плохие практики. «Более агрессивные» настройки в моём понимании такие:

Перевести «предупреждения» в разряд «ошибок»

Предупреждения линтера — это коллективный опыт индустрии, который может оказаться ценным. Однако предупреждения легко пропустить, потому что они не заставляют сборку кода «падать с ошибкой».

Если перевести предупреждения в разряд ошибок, то код «перестанет компилироваться». Ошибки будут принуждать «чинить» код или менять правила линтера, которые мы используем.

Не все практики, предлагаемые линтером, могут быть одинаково полезными. Мы можем выбирать правила, которые мы считаем действительно важными, и отмечать другие. Главное, выбрав набор правил, следовать им без отклонений — именно с этим помогает перевод «предупреждений» в «ошибки».

Настроить больше автоматизированных правил

Если команде хочется добавить новые правила для линтера или других автоматизированных инструментов, то это отличный момент попробовать.

Но стоит помнить, что не все правила одинаково полезны и уместны в каждом проекте. Я стараюсь не идти поперёк голоса команды, и если какое-то правило линтера другие разработчики считают ненужным, я не стану его вводить.

К СЛОВУ

О полезных характеристиках кода, которые мне кажутся обязательными и которые при этом можно поймать линтером, мы поговорим отдельно.

-
1. "Working Effectively with Legacy Code" by Michael C. Feathers,
https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
 2. "Debug It! Find, Repair, and Prevent Bugs in Your Code" by Paul Butcher,
<https://www.goodreads.com/book/show/6770868-debug-it>

Во время рефакторинга

После того, как мы обозначили границы изменений, исследовали код и покрыли его тестами, мы можем приступить к рефакторингу.

Во время работы нам хочется, чтобы изменения кода были максимально полезными и при этом находились внутри обозначенных границ. В этой главе обсудим эвристики, которые помогают это делать.

Двигаться маленькими шагами

Маленький шаг — это минимальное изменение, несущее смысл. Хороший пример маленького шага — это *атомарный коммит* (*Atomic Commit*). Такой коммит содержит осмысленное изменение функциональности и переводит кодовую базу из одного рабочего состояния в другое рабочее состояние. С их помощью мы можем развивать и менять кодовую базу, держа её при этом валидной в каждый момент времени.

К слову

Атомарные коммиты позволяют в будущем проводить бисекцию репозитория во время поиска багов. Когда код валиден в каждом из коммитов, мы можем «путешествовать во времени» по репозиторию, переключаясь между разными коммитами.

Размер коммитов зависит от привычек и предпочтений команды. Лично я придерживаюсь правила «чем меньше коммит — тем лучше». Например, в своих проектах я коммичу отдельно даже переименование методов и переменных.

Маленькие шаги побуждают декомпозировать задачи на более простые. Так неподъёмные фичи превращаются в наборы компактных задач, которые не так страшно чаще сливать в основную ветку репозитория. Без декомпозиции такая задача может превратиться в долгострой, блокирующий всю команду.

Суть *частой интеграции в основную ветку* (*Continuous Integration*) в том, чтобы команда как можно чаще синхронизировала изменения в коде между собой. О пользе подхода хорошо написали Марк Симанн в “Code That Fits in Your Head” и Скотт Бернштайн в “Beyond Legacy Code”. ☀ ☀

Блокирующих задач лучше избегать в принципе, а при рефакторинге кода особенно. Если вся команда занята рефакторингом, это может дорого стоить. Прецедент дорогоизны может в будущем лишить проект ресурсов на рефакторинг вообще.

Кроме этого маленькие шаги позволяют в любой момент «отложить» рефакторинг и переключиться на другую задачу. Если мы работаем с `git`, то можем использовать «полочки», чтобы сохранять недоделанную работу через `git stash`. Так процесс разработки будет более гибким, и мы сможем быстрее реагировать на «более срочные задачи» типа внезапных багов на продакшене.

Также результаты маленького шага проще окинуть взглядом и проверить перед тем, как сделать с ними коммит. Такие проверки помогают отсеивать изменения, не относящиеся к текущей задаче. Например, случайные изменения, вызванные автоматическими инструментами типа форматеров или линтеров.

И напоследок, маленькие шаги проще описывать в сообщениях к коммитам. Скоуп таких изменений укладывается в одно-два предложения, поэтому легче описать их смысл коротко и точно.

Создавать маленькие, но подробные пул-реквесты

Этот раздел вытекает из предыдущего, но я хочу заострить на нём внимание отдельно. Проблема больших пул-реквестов в том, что...

Никто не проверяет большие и непонятные пул-реквесты

Если мы хотим улучшить код, то нам нужно, чтобы пул-реквест *проверили* на ревью, тогда наша задача — облегчить работу ревьюерам. Для этого мы можем:

- Ограничить размер изменений — на компактные пул-реквесты проще выделить время и вникнуть в их суть. Ревью не будет выглядеть большой внезапной работой.
- Описать контекст задачи в сообщении к PR. Причины, цель и ограничения задачи помогут поделиться с ревьюерами тем, что известно нам, но ещё не известно им. Так мы сможем предугадать вероятные вопросы и заранее ответить на них — это ускорит ревью.

Пул-реквест — это часть коммуникации в команде. Подробнее о том, как упростить коммуникацию писал Максим Ильяхов в «Новых правилах деловой переписки».⁹

Стремление к маленьким, но подробным PR помогает дробить большие задачи на задачи поменьше и продвигать рефакторинг маленькими шагами.

Проверять каждое изменение

Чтобы код эволюционировал через валидные состояния, мы будем проверять тестами *каждое* изменение, каким бы маленьким оно ни было.

При использовании юнит-тестов можно держать окно с запущенными тестами рядом с редактором.

При использовании более долгих тестов (например, E2E) можно запускать их перед коммитом (например, на pre-commit хуке), чтобы в репозиторий не попадал невалидный код.

Тесты должны побуждать нас коммитить в репозиторий только валидный код. Тогда в каждом коммите будет находиться набор *законченных и осмысленных* изменений.

Это может работать только в том случае, когда мы доверяем тестам. Поэтому в прошлой главе мы и делали акцент на эдж-кейсах и конкретизации результата — они помогают сделать тесты более надёжными.

Применять по одной технике за раз

Частые коммиты фиксируют опорные точки в эволюции кода. Чем такие точки чаще, тем компактнее изменения между ними. Это, например, полезно при осмотре изменений с последнего коммита, которые мы только что внесли. Компактные изменения быстрее изучить, проще понять и не так жалко откатывать.

Во время рефакторинга не всегда очевидно, как делать изменения компактнее и чаще. Мне в этом помогает правило:

Не смешивать разные техники рефакторинга в одном коммите

Это правило помогает коммитить ритмичнее и чаще: переименовали функцию — коммит, вынесли переменную — коммит, добавили код для будущей замены — коммит, и так далее.

Пока мы не смешиваем разные техники в одном коммите, нам проще отслеживать изменения кода по диффам и находить ошибки типа конфликтов имён.

Сложные техники можно разбивать на отдельные этапы, каждый из которых оформлять в виде коммита. В этом случае этапы важно выделять так, чтобы каждый из них тоже оставлял код в валидном состоянии.

К СЛОВУ

Сочетание атомарных коммитов, непрерывной интеграции кода и предкоммитных проверок изменений ещё называют «тактической» работой с гитом. ☈ Мы тоже иногда будем использовать этот термин, говоря о таком применении гита в следующих главах.

Не добавлять фич, не чинить багов

Во время рефакторинга мы можем найти кусок кода, который работает неправильно. Может возникнуть желание «исправить это по пути», но багфиксы и новые фичи к рефакторингу лучше не примешивать.

Рефакторинг *не должен менять функциональность* кода. Если мы добавили фичу во время рефакторинга, и её потребуется откатить, то нам придётся переносить конкретные коммиты или даже строчки кода руками.

Вместо этого все найденные идеи для фич лучше положить в отдельный список и вернуться к ним после рефакторинга. Если мы нашли баг, то рефакторинг стоит отложить (`git stash`) и вернуться к нему после фикса. Для такой манёвренности опять же удобнее работать маленькими шагами.

Соблюдать приоритет преобразований

Приоритет преобразований (*Transformation Priority Premise*, TPP) — это список действий, которые помогают развивать код от наивной простейшей реализации до более сложной, которая отвечает всем требованиям проекта. ☈

TPP помогает *не пытаться сделать всё сразу*. Он побуждает обновлять кусок кода постепенно, записывая каждый шаг в системе контроля версий, и чаще интегрировать изменения в основную ветку репозитория.

Мне лично TPP помогает не перегружать голову деталями, пока я пишу код. Все идеи для улучшений, которые возникают по ходу работы, я складываю в отдельный список. Этот список я потом сравниваю

с TPP и выбираю, что реализовать следующим шагом.

ЗАЧЕМ

«Выгруженные» из головы детали освобождают «оперативную память» мозга. ☺
Высвобожденные ресурсы можно потратить на детали задачи — это улучшает внимательность.

Не смешивать рефакторинг тестов и кода приложения

Тесты и продуктовый код страхуют друг друга. Тесты проверяют, что мы не допустили ошибок в коде приложения, и наоборот. Если рефакторить их одновременно, вероятность пропустить ошибку становится выше.

Рефакторить код приложения и тесты лучше по очереди. Если во время рефакторинга приложения мы заметили, что нужно отрефакторить тест, то стоит:

- «Положить на полочку» изменения в коде приложения;
- Отрефакторить нужный тест;
- Проверить, что тест ломается по той причине, по которой должен;
- «Достать с полочки» предыдущие изменения и продолжить работать над ними.

ПОДРОБНЕЕ

Чуть подробнее об этой технике мы поговорим в главе о рефакторинге тестового кода.

-
1. Atomic Commit, Wikipedia https://en.wikipedia.org/wiki/Atomic_commit
 2. git-bisect, Use binary search to find the commit that introduced a bug, <https://git-scm.com/docs/git-bisect>
 3. “Write Better Commits, Build Better Projects” by Victoria Dye, <https://github.blog/2022-06-30-write-better-commits-build-better-projects/>
 4. “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
 5. “Beyond Legacy Code” by David Scott Bernstein, <https://www.goodreads.com/book/show/26088456-beyond-legacy-code>
 6. «Новые правила деловой переписки» М. Ильяков, Л. Сарычева, <https://www.goodreads.com/book/show/41070833>
 7. “Use Git Tactically” by Mark Seeman, <https://stackoverflow.blog/2022/04/06/use-git-tactically/>

8. Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise

9. Оценка емкости рабочей памяти, Википедия, <https://ru.wikipedia.org/wiki/>

[Рабочая память#Оценка емкости рабочей памяти](#)

Низко-висящие фрукты

Для рефакторинга куска кода могут потребоваться разные изменения, и бывает сложно выбрать, с чего начать. Мне нравится ранжировать изменения от простых к сложным и начинать с простых. Это помогает «сдуть с кода пыль» и начать видеть в нём более серьёзные проблемы.

К простым изменениям я обычно отношу форматирование, ошибки линтера и замену самописного кода на возможности языка или окружения, в котором код будет работать. Об этом сейчас и поговорим.

Форматирование кода

Форматирование — это вкусовщина, но у него есть одна полезная функция. Если код во всём проекте написан *последовательно*, то у читателей уходит меньше времени на его восприятие.

Так работают привычки: знакомый «рисунок» кода помогает сосредоточиться не на словах и буквах, а на смысле.

```
// Код без форматирования.  
// При чтении надо «продираться» сквозь него,  
// чтобы увидеть смысл за буквами:  
  
function ProductList({ products }) {  
    return <ul>{products.map((product) =><li key={product.name}>  
        <Product product={product} /></li>)}</ul>  
  
// Отформатированный код помогает упростить чтение  
// и быстрее перейти к смыслу:  
  
function ProductList({ products }) {  
    return (  
        <ul>  
            {products.map((product) => (  
                <li key={product.name}>  
                    <Product product={product} />  
                </li>  
            ))}  
        </ul>  
    );  
}
```

Форматирование лучше автоматизировать. В примере выше я использовал Prettier, [♂] но конкретный инструмент здесь не так важен, как подход в целом. Если команду не устраивает Prettier, можно выбрать другой форматер и использовать его. Суть в *автоматизации процесса*.

Бывает, однако, что форматер ломает работу кода, например, если не учитывает особенности ASI (Automatic Semicolon Insertion) [♂] в JavaScript:

```
// До форматирования:

export function connect(userChannel, appChannel, credentials) {
  establishConnection(credentials)
  [userChannel, appChannel].forEach(handshake)
}

// После:

export function connect(userChannel, appChannel, credentials) {
  establishConnection(credentials)[(userChannel, appChannel)].forEach(
    handshake
  );
}
```

Мы можем заметить эту ошибку сами, если пользуемся гитом «тактически» и проверяем, что именно изменилось с последнего коммита:

```
4
5 export function connect(userChannel,
6   appChannel, credentials) {
7 - establishConnection(credentials)
8 - [userChannel, appChannel].forEach(handshake)
9 }
10

4
5 export function connect(userChannel,
6   appChannel, credentials) {
7+ establishConnection(credentials)
8+ [(userChannel, appChannel)].forEach(
9+   handshake
10 );}
```

Гит показывает, к каким изменениям привело форматирование

...Но искать такие ошибки только лишь вручную ненадёжно, поэтому их поиск тоже лучше автоматизировать.

Надёжнее всего покрыть код тестами перед началом рефакторинга. Если мы запустим тесты в интерактивном режиме рядом с редактором, то будем сразу же видеть, какие ошибки появились после применения форматера. Тогда полагаться только на ручную проверку работы автоматических инструментов понадобится реже.

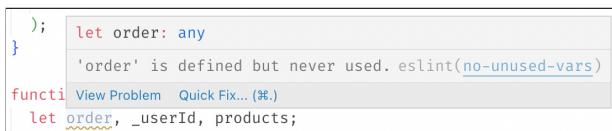
Применение форматирования можно считать отдельной техникой рефакторинга, поэтому результат можно оформить в виде коммита или даже отдельного PR. Наша задача здесь как можно раньше интегрироваться в основной веткой, чтобы не приходилось разруливать сложные конфликты между обновлением форматирования от нас и смысловыми изменениями кода от других разработчиков.

Линтинг кода

Включив линтер и переведя «предупреждения» в «ошибки», мы можем получить список таких новых ошибок. Этот список можно использовать как список задач для текущей итерации рефакторинга.

Мне нравится оформлять работу над каждым из правил линтера как отдельный коммит или PR.

Например, можно удалить весь неиспользуемый код, оформить это как коммит и перейти к следующей проблеме из списка.



```
);  
}  
let order: any  
'order' is defined but never used. eslint(no-unused-vars)  
functi View Problem Quick Fix... (#.)  
let order, _userId, products;
```

Линтер подсвечивает неиспользуемый код, который можно удалить

Если ошибок после включения линтера очень много, то можно включать не все правила сразу, а по одному. Чем мельче будут шаги, тем проще распилить задачу на несколько и решить каждую отдельно.

После исправления каждого правила потребуется проверить, не сломались ли тесты. В будущем я перестану акцентировать внимание на проверке тестов, чтобы сократить текст. Просто договоримся держать в голове, что мы проверяем, не сломались ли тесты, после каждого изменения.

Возможности языка

Современные языки программирования развиваются и получают обновления. Особенно это применимо к JavaScript, так как спецификация ES обновляется каждый год. ☀

Иногда в новой версии языка появляются фичи, которыми можно заменить старые самописные функции. Как правило, встроенные конструкции языка компактнее, быстрее, надёжнее и понятнее. Мы можем внедрять новые возможности языка, оглядываясь на требуемую поддержку с помощью, к примеру, Canuse. ☀

JAVASCRIPT

```
// Самодельный хелпер для проверки начала строки:  
const startsWith = (str, chunk) => str.indexOf(chunk) === 0;  
const yup = startsWith("Some String", "So");  
  
// ...Можно заменить нативным методом:  
const yup = "Some String".startsWith("So");
```

К СЛОВУ

Если самописная реализация отличается от нативной, и мы не можем заменить её, то я предпочитаю отметить это в документации. Так будет понятно, почему мы используем свои наработки вместо возможностей языка.

Возможности встроенных API

Кроме возможностей языка полезно помнить о функциях стандартной библиотеки и доступных встроенных API. Правильно подобранный функция или структура данных поможет сделать код эффективнее, чище и короче. ↗ ↗ Например, во фрагменте ниже мы можем упростить самописную сериализацию формы с помощью `FormData`:

JAVASCRIPT

```
// Многословный ручной обход всех полей со значениями:  
const username = form.querySelector('[name="username"]').value;  
const password = form.querySelector('[name="password"]').value;  
const data = { username, password };  
  
// ...Мы можем заменить на вызов стандартного API:  
const data = Object.fromEntries(new FormData(form));
```

Количество кода после изменений уменьшилось, а сам код стал устойчивее к изменениям и расширению функциональности. Например, после рефакторинга нам не потребуется вручную обновлять сериализацию при добавлении нового поля в форму:

JAVASCRIPT

```
// Раньше нам бы пришлось обновлять объект руками:  
// ...  
const email = form.querySelector('[name="email"]').value;  
const data = { username, email, password };  
  
// Сейчас же `FormData` дополнит данные самостоятельно,  
// обновлять код вручную не потребуется.
```

Удалять код выгодно: чем меньше кода, тем меньше потенциальных точек отказа в работе приложения. В целом, при прочих равных я предпочту отдать большую часть работы языку или окружению, чтобы не писать код самостоятельно. Так обычно получается надёжнее.

Иногда мы не можем решить задачу, используя только стандартные API, и мы вынуждены изобретать велосипед. В таких случаях также стоит зафиксировать в документации причины, почему стандартные решения не подходят.

Подробнее о том, как сделать документацию и комментарии к коду информационно насыщенными и полезными, мы поговорим в одной из следующих глав.

Возможности среды разработки

Вместе с возможностями языка ещё хочется выделить возможности редактора или IDE, с которыми мы работаем. Если в них есть автоматизированные средства рефакторинга — стоит научиться ими пользоваться.

“Rename Symbol”, “Extract into Function” и другие инструменты ускоряют работу и снижают когнитивную нагрузку. Например, в VS Code можно изменить имя функции или переменной во всех местах использования сочетанием горячих клавиш: ⌘

```

const user = { name: "Alex" };
const wallet = { cards: [] };

const account = { ...user, ...wallet };

const user = { name: "Alex" };
const profile
  Enter to Rename, ⌘Enter to Preview
const account = { ...user, ...wallet };

const profile = { name: "Alex" };
const wallet = { cards: [] };

const account = { ...profile, ...wallet };

```

“Rename Symbol” обновляет название сразу и везде

Однако, результат применения этих инструментов стоит перепроверять. Например, Rename Symbol может «не заметить» какое-то имя или добавить лишнее переименование:

```

// Например, мы хотим заменить поле 'name'
// в типе 'AccountProps' на 'firstName':

type AccountProps = { name: string };
const Account = ({ name }: AccountProps) => <>{name}</>

// После применения Rename Symbol
// может остаться «лишнее переименование»:

type AccountProps = { firstName: string };
const Account = ({ firstName: name }: AccountProps) => <>{name}</>;

```

Чтобы этого избежать, мы снова можем воспользоваться преимуществами «тактической» работы с гитом. Для этого пробежимся по изменениям с последнего коммита и проверим, что именно переименовалось и как:

1 type AccountProps = { 2 - name: string; 3 }; 4 5- const Account = ({ name }: - AccountProps) => (6 <main> 7 <h1>{name}</h1> 8 </main> 9);	1 type AccountProps = { 2 + firstName: string; 3 }; 4 5+ const Account = ({ firstName: name }: + AccountProps) => (6 <main> 7 <h1>{name}</h1> 8 </main> 9);
--	--

Гит показывает, какие поля объекта затронуло использование "Rename Symbol"

Упростить и максимизировать пользу от такого сравнения помогает стратегия маленьких шагов, о которой мы говорили в предыдущей главе. Если применять лишь одну технику рефакторинга за коммит, в диффах не будет шума и будет лучше видно, как именно изменения повлияют на код.

Линтеры и тесты при этом помогут избежать конфликтов имён и других ошибок. Например, мы можем настроить правила, которые запретят одинаковые имена переменных, и тогда при конфликте имён линтер будет падать с ошибкой. Если он запущен параллельно с редактором, то мы это сразу же увидим и сможем исправить.

1. Prettier, an opinionated code formatter, <https://prettier.io>

2. “Automatic semicolon insertion in JavaScript” by Dr. Axel Rauschmayer, https://2ality.com/2011/05:semicolon_insertion.html

3. List of EcmaScript Proposals, <https://proposals.es>

4. Can I Use, support tables for web, <https://caniuse.com>
5. "Use the Right Algorithm and Data Structure" by JC van Winkel, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_89/
6. "How to Convert HTML Form Field Values to a JSON Object" by Jason Lengstorf, <https://www.learnwithjason.dev/blog/get-form-values-as-json>
7. Refactoring Source Code in VSCode, <https://code.visualstudio.com/docs/editor/refactoring>

Имена

После того, как мы «сдули с кода пыль», мы готовы рассматривать более серьёзные проблемы.

Изменения и техники, о которых мы будем говорить дальше, значительно труднее ранжировать от простых к сложным. Не всегда очевидно, когда и какое изменение необходимо применить следующим.

ПОРЯДОК ГЛАВ

Я предлагаю читателям воспринимать эту и следующие главы не как «пошаговый мануал», а скорее как справочник со списком проблем и их возможных решений.

Главы в книге расположены так, что размер предлагаемых изменений постепенно растёт. Но нет гарантий, что при рефакторинге реального проекта проблемы будут встречаться именно в таком порядке.

Будьте внимательны при анализе кода. Подмечайте и выписывайте проблемы так, как вам удобнее всего, а эту книгу используйте как вспомогательный материал.

Внимание к именам переменных, функций, классов и модулей может стать крышкой ящика Пандоры в мире рефакторинга. «Непонятные» имена могут быть сигналом о недостаточном разделении ответственности между модулями, высоком зацеплении кода, неадекватном отражении предметной области или «протекающих» абстракциях. В этой главе поговорим о том, как имена помогают искать проблемы и на что обращать внимание при рефакторинге кода с непонятными именами.

Общие рекомендации

За «непонятным» именем могут скрываться ценные детали проекта и опыт прошлых разработчиков. Нам важно сохранить их, поэтому стоит погрузиться в смысл «непонятного имени». Если во время рефакторинга мы видим имя, смысл которого нам не ясен, то можно попробовать:

- найти кого-то в команде, кто знает значение этого имени;

- найти документацию к коду, чтобы узнать смысл оттуда;
- в крайнем случае провести серию экспериментов, меняя значение переменной, и смотря, как это влияет на работу кода.

К СЛОВУ

Если влияние переменной можно засечь на «шве», то с экспериментами помогут тесты. Их результаты покажут, как меняется работа кода в зависимости от различных значений этой переменной, что поможет сделать предположение о её сути.

В остальных случаях оценивать влияние на работу кода, вероятно, придётся вручную. Один из вариантов оценки — понаблюдать в отладчике, как изменения отражаются на других переменных и данных, с которыми работает код.

Такие эксперименты не дадут *гарантий*, что мы поймём смысл сущности правильно, \wp но смогут подсказать, каких знаний о проекте нам не хватает.

Собранные знания лучше отразить прямо в коде, переименовав переменную. Если это невозможно, то стоит аккумулировать их *как можно ближе к коду*. Это может быть комментарий, документация, сообщение о коммите, описание задачи в трекере. Суть в том, чтобы дать разработчикам возможность найти эти сведения и обратиться к ним, *тратя как можно меньше умственных усилий и времени*.

Слишком короткие имена

Имя — хорошее, если оно адекватно отображает информацию о предметной области или смысле переменной. Чрезмерно короткие имена и необщепринятые сокращения этого не делают. Рано или поздно такое имя будет прочитано неправильно, потому что все «знающие» разработчики перестанут работать над проектом.

```
// Имена `d`, `c` и `p` слишком короткие,
// чтобы сделать вывод о смысле переменных:

let d = 0;
if (c === "HAPPY_FRIDAY") d = p * 0.2;

// Если мы назовём переменные полными словами,
// то понять смысл операции станет проще:

let discount = 0;
if (coupon === "HAPPY_FRIDAY") discount = price * 0.2;
```

Однобуквенные имена безобидны, например, в циклах или очень коротких кусочках кода. Но для бизнес-логики, где используются термины из предметной области, лучше использовать полные формы этих терминов.

То же относится к сокращениям. Я предполагаю не использовать сокращения в коде, кроме 2 случаев:

- если сокращение общезвестное (USA, OOP, USD);
- если оно используется в предметной области именно в таком сокращённом виде (Challenge Rate как CR в калькуляторе монстров для D&D).

В остальных случаях я предполагаю писать имя для переменной в полном виде:

```
// Всё в порядке, сокращение USD общезвестное:
const usd = {};  
  
// Может быть приемлемо, если предметная область программы
// связана с математикой, а модуль работает с производными:
const dx = 0.42;  
  
// Не лучший вариант, лучше расшифровать:
const ec = 0.6188;  
  
// Хорошо, так понятнее:
const elasticityCoefficient = 0.6188;
```

Слишком длинные имена

Слишком длинные имена намекают, что сущность делает чрезмерно много разных дел. Ключевое слово тут *разных*, потому что именно разношёрстную функциональность сложнее всего объединить в одном имени.

Когда функциональность слабо связана по смыслу, имя старается передать весь контекст работы в одной фразе. Это раздувает имя, делает его шумным. Один из сигналов обратить внимание на имя — это наличие в нём слов типа `that`, `which`, `after` и т.д.

Чаще всего длинными именами «болеют» функции, которые делают слишком много. Такие функции пытаются объясняться терминами, которые для них либо слишком примитивны, либо наоборот — слишком абстрактны, и им приходится искать подходящие слова. Моя главная эвристика для поиска таких функций такова: если я читаю код функции и не могу придумать имя покороче, скорее всего, она делает слишком много.

JAVASCRIPT

```
async function submitOrderCreationFormIfValid() {  
    // ...  
}
```

Функция `submitOrderCreationFormIfValid` из примера выше делает сразу 3 вещи:

- обрабатывает отправку формы;
- валидирует данные из неё;
- создаёт заказ.

Вероятно, каждый шаг достаточно важный, чтобы отразить его в имени, поэтому оно очень большое. Но лучше подумать, как разделить задачу на задачи поменьше и разделить ответственность между отдельными функциями:

```
// Сериализует форму в объект
// или другую удобную для работы структуру:
function serializeForm() {}

// Валидирует данные из формы:
function validateFormData() {}

// Создаёт объект заказа по собранным данным:
function createOrder() {}

// Отправляет заказ на сервер:
async function sendOrder() {}

// Реагирует на DOM-событие, вызывая другие функции:
function handleOrderSubmit() {}
```

Тогда вместо одной большой функции, которая пытается делать всё, мы бы получили цепочку из нескольких, действия внутри которых были бы *связаны по смыслу*. Их имена могли бы отражать детали работы корневой функции, что облегчило бы и её имя:

```
async function handleOrderSubmit(event) {
  const formData = serializeForm(event.target);
  const validData = validateFormData(formData);
  const order = createOrder(validData);
  await sendOrder(order);
}
```

Кажется, что мы так ухудшаем точность имени корневой функции, ведь мы убираем из него детали. Но всё дело в том, какие именно детали мы убираем.

Хорошая тактика — думать об имени функции с точки зрения кода, который будет эту функцию вызывать. Важно ли форме, как именно обработают её отправку? Вероятно нет; ей важен факт обработки и чтобы ей занималась нужная функция:

- Обработать handle +
- отправку submit +
- формы заказа order
-
- handleOrderSubmit

А вот как именно это будет происходить, важно уже самой функции-обработчику

`handleOrderSubmit`. Детали реализации важны только внутри функции, но не важны в её имени.

Внутри же эти детали можно выразить через имена внутренних функций.

К слову

Вероятно, вы узнали в этом пример разделения уровней абстракции.  Подробнее об этом мы поговорим в отдельной главе.

Основное решение проблем слишком длинных имён — посмотреть, что именно имя пытается рассказать. Мы можем постараться вытащить из имени все детали, которые оно несёт, а затем разделить их на «важные снаружи» и «важные внутри». Это помогает разделить задачу на более простые.

ПРИ ЭТОМ

Логика функции сама по себе вполне может быть сложной. Просто если её «внутренности» собраны в «кушки по смыслу», проблем с именованием, как правило, возникает меньше.

Вариант именования функций

Если говорить о функциях, то лавировать между «слишком короткими» и «слишком длинными» именами помогает шаблон A/HC/LC.  Он предлагает сочетать в себе само действие, кто его совершает или над чем его совершают:

`prefix? + action (A) + high context (HC) + low context? (LC)`

Его можно использовать как опорную или стартовую точку при именовании функций, изменяя имя по мере необходимости.

Однаковые имена у разных сущностей

Во время рефакторинга также стоит обратить внимание на ощущение «путаницы» в коде. Оно может возникнуть, когда разные явления или предметы в коде описаны одинаковыми именами.

При чтении мы в первую очередь опираемся на имена классов, функций и переменных, чтобы понять смысл кода. Если имя не соотносится с переменной 1 к 1, нам приходится каждый раз вспоминать, о какой именно переменной идёт речь. Для этого нам нужно держать в голове «мета-информацию» об этой переменной и каждый раз обращаться к ней. Это делает чтение сложнее.

В некоторых случаях это же относится и к разным именам для одной переменной. Например, лучше избегать разных имён для доменных сущностей, но об этом чуть позже.

Чтобы избежать высокой когнитивной нагрузки при чтении, лучше для *разных* сущностей использовать *разные* имена:

JAVASCRIPT

```
// Здесь `user` – это объект с данными пользователя:
function isOldEnough(user, minAge) {
    return user.age >= minAge;
}

// А тут – имя:
function findUser(user, users) {
    return users.find(({ name }) => name === user);
}

// С первого взгляда их сложно различить,
// потому что имена «намекают»,
// будто переменные значат одно и то же.

// Это может вводить в заблуждение; нам придётся помнить,
// что внутри функции `findUser` переменная `user`
// относится не к объекту пользователя, а к имени.

// Лучше выразить смысл переменной точнее прямо через имя:
function findUser(userName, users) {
    return users.find(({ name }) => name === userName);
}
```

Однаковые имена у разных сущностей особенно вредят, если они находятся рядом или используются близко друг с другом. Общий контекст лишь усиливает ощущение, что одинаковое имя отвечает за одинаковый смысл.

ОДНАКО

Из этого правила, конечно, есть исключения. Например, одно и то же имя может использоваться для разных целей, если код страхует систему типов, или если из контекста использования понятно, о чём именно идёт речь. Но по умолчанию лучше использовать разные имена.

Повсеместный язык

Бороться с проблемой именования помогает **повсеместный язык** (*Ubiquitous Language*). Это набор терминов, описывающих предметную область, которыми пользуется вся команда. Под «всей командой» мы имеем в виду не только команду разработки, а всего продукта в целом, включая дизайнеров, продукт-онеров, стейхолдеров и т.д.

К СЛОВУ

Сам термин пришёл из методологии *предметно-ориентированного проектирования* (*Domain-Driven Design, DDD*).[♂] Мне она кажется удобной для описания предметной области, возможно, вам она тоже окажется полезной.

На практике это значит, что если «люди из бизнеса» для описания заказов используют термин «Заказ» (*Order*), то именно этим словом мы и будем называть заказы в коде, тестах, документации и устной коммуникации.

Сила повсеместного языка в *однозначности*. Если все называют вещь одним и тем же именем, то потерю при «переводе с языка бизнеса на язык разработки» будет меньше.

ПОДРОБНЕЕ

Подробнее об этом писал Скотт Влашин в книге “Domain Modelling Made Functional”, очень рекомендую к прочтению.[♂]

Когда мы выражаемся понятиями, близкими к реальности, ментальная модель программы становится ближе к тому, что мы моделируем. Неправильное поведение программы в этом случае заметить гораздо проще.

Врущие имена

Иногда во время рефакторинга находятся имена, которые неадекватно отражают суть переменной или функции. Они могут быть очень неточными или вовсе врать. Такие имена лучше исправить как можно раньше.

Если мы не уверены, как будет правильнее назвать переменную, то стоит выписать причины, почему имя не подходит. Например, если мы встретили вот такой код:

```
const trend = currentValue - previousValue;

// «Тренд» описан как разница между «Текущим» и «Предыдущим»
// значением некоторой характеристики.
```

...А в разговорах мы замечали, что в команде это называют «Дельтой» (а не «Трендом»), то стоит выписать это замечание в список. Получится что-то типа:

- «Тренд», вероятно, описывает суть не точно;
- Устно чаще используется термин «Дельта», в том числе — продукт-оунерами;
- Специфика проекта, возможно, требует, чтобы тренд строился по более, чем двум точкам.

Эти опасения можно представить команде, чтобы подумать над переименованием.

Если имя очевидно врёт, то этап обсуждения можно пропустить. Но если есть сомнения «Переименовывать ли?», то стоит сперва их обсудить с другими разработчиками и продукт-оунером.

Типы для описания домена

В языках со статической типизацией мы можем использовать типы, как способ передачи деталей предметной области. Это снимает нагрузку с имени сущности, вынося некоторые детали в тип. Имя становится лаконичнее, но из-за типа почти не теряет смысла.

ОДНАКО

Можно поспорить, сказав, что тип переменной видно только в сигнатурах или при наведении на неё, а имя видно всегда.

Но, во-первых, большое количество длинных названий будет шуметь, и ценность деталей исчезнет. А, во-вторых, в тип я обычно выношу детали, которые не нужны *мгновенно* — за такими деталями можно обратиться к подсказке в IDE.

Мне кажется, что такой компромисс вполне допустим.

К примеру, типами удобно описывать состояния, которые проходят данные на разных этапах работы приложения:

```

type CreatedOrder = {
    createdAt: TimeStamp;
    createdBy: UserId;
    products: ProductList;
};

type ProcessedOrder = {
    createdAt: TimeStamp;
    createdBy: UserId;
    products: ProductList;
    address: Delivery;
};

```

Мы таким образом не только описываем разные состояния данных (по сути разные сущности) отличающимися именами, но и запрещаем использовать «неправильный» тип там, где он не подходит. Например, можно запретить попытки отправить неподготовленные заказы:

```

function sendOrder(order: ProcessedOrder) {
    // ...
}

const order: CreatedOrder = {
    /*...*/
};

// Вызов функции ниже не скомпилируется,
// потому что тип аргумента не подходит под сигнатуру функции.
// Такую сигнатуру можно воспринимать как предупреждение:
// «заказ ещё не готов к тому, чтобы пытаться его отправить».
sendOrder(order);

```

ПОДРОБНЕЕ

О том, «почему просто не использовать Boolean-флаги для разных состояний», мы поговорим подробнее в главе о статической типизации.

1. «После» не значит «вследствие», Википедия, https://ru.wikipedia.org/wiki/Логическая_ошибка#Мнимая_логическая_связь
 2. Уровни абстракции, Википедия, [https://ru.wikipedia.org/wiki/Уровень_абстракции_\(программирование\)](https://ru.wikipedia.org/wiki/Уровень_абстракции_(программирование))
 3. Шаблон именования A/HC/LC, <https://github.com/kettanaito/naming-cheatsheet#ahclc-pattern>
 4. “Domain-Driven Design” by Eric Evans,
https://www.goodreads.com/book/show/179133.Domain_Driven_Design
 5. “Domain Modeling Made Functional” by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>

Дублирование кода

Главная цель рефакторинга — сделать код более читабельным. Один из способов этого достичь — уменьшить в нём количество шума.

Дублирование кода шумит, когда не несёт в себе полезной информации. Однако, далеко не всякое дублирование — зло. Оно может быть инструментом разработки и проектирования, поэтому при рефакторинге нам стоит понимать, с каким именно дублированием мы имеем дело.

В этой главе мы обсудим, на что обращать внимание при выявлении дублирования и как понимать, когда от него пора избавляться.

Не любое дублирование — зло

Если у двух кусков кода одинаковая цель, они содержат одинаковый набор действий и работают с одинаковыми данными — это *прямое дублирование*. От него можно смело избавляться, например, выделив повторяющийся код в переменную, функцию или модуль.

Но бывает, что две части кода «вроде похожи», а спустя время оказываются совершенно разными. Если поспешить и объединить их слишком рано, то распиливать такой код будет сложнее, чем объединять действительно одинаковый код позже.

Когда мы не уверены, что перед нами два *действительно одинаковых* куска кода, мы можем отметить эти места специальными метками и добавить предположение о том, что в них дублируется.

JAVASCRIPT

```
/** @duplicate Применяет купон на скидку к заказу. */
function applyCoupon(order, coupon) {}

/** @duplicate Применяет купон на скидку к заказу. */
function applyDiscount(order, discount) {}
```

Такие метки принесут пользу, только если проводить их регулярные аудиты. Во время аудитов нам следует проверять, что нам стало известно нового о возможных дубликатах.

К СЛОВУ

Регулярные аудиты в своих проектах я воспринимаю как часть выплаты технического долга. Для подобных периодических задач я завожу списки дел. Внутри списка я указываю, что надо сделать в рамках той или иной задачи. Техника регулярных аудитов и её польза хорошо описана у Максима Дорофеева в «Джедайских техниках».[♂]

Если во время аудита метки стало ясно, что описанное в ней дублирование *прямое*, мы можем провести рефакторинг кода с этой меткой. Если же код оказался разным, метку можно удалить. Это помогает не спешить с обобщением кода, но при этом не терять места вероятного дублирования.

Переменные для данных

Дублирование статических данных или результатов вычислений удобно выносить в переменные или наборы переменных. Это, например, помогает в распутывании сложных условий или выделении этапов преобразований данных.

JAVASCRIPT

```
// Если условия расположены близко,  
// или используются рядом:  
  
if (user.age < 18) toggleParentControl();  
// ...  
if (user.age < 18) askParents();  
  
// Мы можем вынести выражение в переменную:  
  
const isChild = user.age < 18;  
  
if (isChild) toggleParentControl();  
// ...  
if (isChild) askParents();
```

Иногда дублирование может быть менее очевидным, и заметить его сложнее:

```
// Второе условие «вывернуто»
// и использует переменную `years` ,
// а не поле объекта напрямую.

if (user.age < 18) askParents();
// ...
const { age: years } = user;
if (years >= 18) askDocuments();

// Но мы всё ещё можем избавиться от него,
// вынеся выражение в переменную:

const isChild = user.age < 18;

if (isChild) askParents();
// ...
if (!isChild) askDocuments();
```

ПОДРОБНЕЕ

Об упрощении и распутывании сложных условий мы детальнее поговорим в одной из следующих глав.

ФУНКЦИИ ДЛЯ ДЕЙСТВИЙ

Повторяющиеся действия или преобразования данных удобно выносить в функции и методы.

Определять дубликаты помогает «проверка на одинаковость»:

- Перед нами прямое дублирование, если у действий одинаковая цель — то есть желаемый результат;
- Однаковая область действия — часть приложения, на которую они влияют;
- Одинаковые прямые входные данные — аргументы и параметры;
- Одинаковые непрямые входные данные — зависимости и импортируемые модули.

В примере ниже фрагменты кода такую проверку проходят:

```
// - Цель: добавить поле со абсолютным значением скидки к заказу;
// - Область: объект заказа;
// - Прямые данные: заказ без скидки, относительное значение скидки;
// - Непрямые данные: конвертер из процентов в абсолютное значение.

// a)
const fromPercent = (amount, percent) => (amount * percent) / 100;

const order = {};
order.discount = fromPercent(order.total, 50);

// b)
const order = {};
const discount = (order.total * percent) / 100;
const discounted = { ...order, discount };

// Действия одинаковые, мы можем вынести их в отдельную функцию:

function applyDiscount(order, percent) {
  const discount = (order.total * percent) / 100;
  return { ...order, discount };
}
```

В другом примере цель и прямые входные данные фрагментов кода одинаковые, а вот зависимости отличаются:

```
// Первый фрагмент считает скидку в процентах,  
// а второй использует «скидку дня» – `todayDiscount`.  
  
// a)  
const order = {};  
const discount = (order.total * percent) / 100;  
const discounted = { ...order, discount };  
  
// b)  
const todayDiscount = () => {  
    // ...Подбор скидки к сегодняшнему дню.  
};  
  
const order = {};  
const discount = todayDiscount();  
const discounted = { ...order, discount };
```

В примере выше у фрагмента "b" среди зависимостей есть функция `todayDiscount`. Из-за неё наборы действий отличаются достаточно, чтобы считать их «похожими», но не «одинаковыми».

Мы можем использовать `@duplicate`-метки и проследить за развитием событий, чтобы получить больше информации о том, как работает предметная область. Когда мы точно знаем и уверены, как должны работать эти фрагменты, мы можем действия «обобщить»:

```
// Обобщённая функция будет принимать
// абсолютное значение скидки:

function applyDiscount(order, discount) {
  return {...order, discount}
}

// Отличия в подсчёте (процент от суммы, «скидка дня» и т.д.)
// соберём в виде отдельного набора функций:

const discountOptions = {
  percent: (order, percent) => order.total * percent / 100
  daily: daysDiscount()
}

// В результате получим обобщённую функцию
// и словарь со скидками разных видов.
// Тогда применение любой скидки станет единообразным:

const a = applyDiscount(order, discountOptions.daily)
const b = applyDiscount(order, discountOptions.percent(order, 40))
```

ПОДРОБНЕЕ

Детально об обобщённых алгоритмах, их использовании и параметризации мы поговорим в отдельной главе.

1. «Джедайские техники» Максим Дорофеев, <https://www.goodreads.com/book/show/34656521>

Абстракция

Код с лишними деталями «шумит», его сложно читать и держать в голове. Шумный код пересыщен информацией и делает так много, что сложно понять, где заканчивается один осмысленный шаг и начинается другой.

Как правило, шумный код описывает сложное действие с большим количеством этапов. Чем сложнее действие, тем больше деталей приходится учитывать, тем больше вспомогательных шагов нужно описать. Сваленные в кучу, эти детали начинают шуметь и перегружать голову читателя.

В этой главе мы обсудим, как уменьшать количество шума в коде с помощью абстракции и выявлять детали, важные «снаружи» и «внутри» функций.

Намерение и реализация

Абстракция — это устранение неважного и усиление существенного 



В главе об именах в одном из примеров мы выделяли в имени функции детали, которые «важны снаружи» и «важны внутри». Так мы упрощали имя и находили баланс между количеством информации в имени функции и её теле. Самые важные детали мы отражали в имени, а менее важные — прятали в реализации. Такое деление деталей «по уровням» — и есть абстракция.

Абстракция помогает укрощать сложность, разделяя *намерение и реализацию*. Намерение описывает, что мы собираемся делать, а реализация — как. Намерение важно на «верхнем» уровне при описании сущности «в общих чертах» и её взаимодействии с окружением. Детали реализации важны на уровне ниже, когда мы фокусируемся на работе этой сущности и её внутренних процессах.

```
// Название и сигнатура функции отражают намерение...
function isChild(user) {
    // ...А тело функции – реализацию.
    return user.age < 18;
}

// Когда мы используем эту функцию с другими,
// нам важны её цель и назначение, а не детали реализации:
if (isChild(user)) toggleParentControl();
```

Наш мозг может работать только с ограниченным количеством информации одновременно. Абстракция помогает фокусироваться на деталях, которые важны сейчас. Необходимость такой фокусировки заметна в коде, детали из «разных уровней» смешаны.

Рассмотрим пример. Представим, что у нас есть функция подписки на рассылку `subscribeToFeed`, которая проверяет валидность почты перед началом работы:

```
function subscribeToFeed(email) {
    if (!email.includes("@") || !email.includes(".")) return false;

    const recipients = addRecipient(email);
    confirmFeedSubscription(recipients);
}
```

Если мы сравним валидацию почты с остальными действиями (`addRecipient`, `confirmFeedSubscription`), то увидим, что она «изъясняется слишком примитивными терминами» для этой задачи.

Пока другие функции говорят о «почте», «подписках» и «ленте», валидация говорит о символах `"@"` и `". ."`. Из-за этого нам будто приходится при чтении «прыгать» между деталями проверки почты и целью этой проверки.

Функция `subscribeToFeed` действительно хочет знать, валиден ли переданный адрес. Но как именно почта будет проверена, этой функции не важно. Отсюда мы можем сделать вывод, что *детали проверки* здесь лишние.

Чтобы решить эту проблему, мы можем вынести (абстрагировать) проверку почты в отдельную функцию `isValidEmail`:

```

function isValidEmail(email) {
  return email.includes "@" && email.includes ".";
}

function subscribeToFeed(email) {
  if (!isValidEmail(email)) return false;

  const recipients = addRecipient(email);
  confirmFeedSubscription(recipients);
}

```

Теперь имя функции `isValidEmail` «упаковывает» целый набор действий в одну фразу. Имя при этом выражается терминами, близкими к тем, что используют имена функций вокруг неё. Такая близость помогает фокусироваться на целях «упакованных» действий, а не их внутренних процессах.

К СЛОВУ

Этого добиться чуть сложнее в коде с сайд-эффектами, но о них мы поговорим в отдельной главе.

Когда мы вызываем функцию `isValidEmail`, мы фокусируемся на её имени и намерении. В этот момент нам важно, как эта функция взаимодействует с сущностями вокруг — «что происходит, если почта не валидна».

Если же нам важны правила проверки, то мы заглянем в тело функции — реализацию. В этот момент нам будет важно, как функция решает, что вернуть: `true` или `false`.

К СЛОВУ

В языках со статической типизацией намерение удобно выражать и через сигнатуру функции. Об этом поговорим подробнее в главе о статической типизации.

Абстракция помогает «погружаться» в сложные понятия и процессы постепенно, выдаёт информацию о системе дозировано. На каждом «уровне детализации» нам доступно ровно столько информации, сколько нужно для понимания процессов на этом уровне. Марк Симанн называет это фрактальной архитектурой, и эта метафора мне кажется полезной при рефакторинге кода.[♂]

Фрактальная архитектура

По сравнению с компьютером наш мозг вычислительно слаб. Нам трудно перемножать в уме большие числа, представлять в деталях сложные схемы или держать в голове одновременно много понятий.

При рефакторинге нам стоит следить за тем, насколько сложно «держать кусок кода в голове». Если приходится напрягаться, чтобы помнить обо всех деталях — в коде есть проблема.

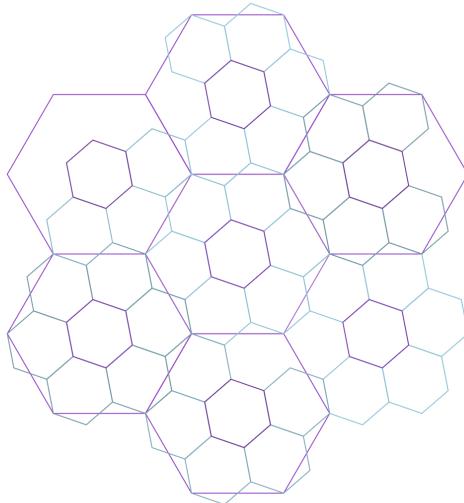
В хорошем коде на экране ровно столько информации, сколько нужно читателю в конкретный момент. Марк Симанн предлагает писать программы так, чтобы на каждом «уровне детализации» количество составных частей не превышало некоторый лимит. Мы можем использовать эту эвристику, чтобы проверять, насколько код пересыщен деталями.[♂]

[К СЛОВУ](#)

В качестве лимита Марк предлагает число 7. Он опирается на предпосылки, что мы можем держать в голове 7 ± 2 объекта.[♂] [♂] При этом он оговаривается, что конкретное число не принципиально, главное — наличие такого лимита.

Для визуализации «уровней» он предлагает использовать сетку из шестиугранников. Каждый шестиугранник — это часть системы, которая может быть детализирована глубже, на такие же шестиугранники.

На каждом из уровней детализации мы будем видеть не больше N составных частей, важных для этого уровня. Если нам требуется узнать, как работает конкретная часть, то мы можем «провалиться» на уровень ниже и увидеть, из чего она состоит.



Так программа разбивается на куски, которые разбиваются на куски, которые разбиваются на куски...

КОПИРАЙТ НОТИС

Картинка сгенерирована инструментом из статьи о Fractal Hex Flowers. ↗

Чтобы понять, как это помогает рефакторить код, рассмотрим пример. Допустим, у нас есть приложение, в котором для залогиненных пользователей мы показываем дашборд, а для остальных — страницу входа.

Входная точка приложения (верхний уровень детализации) может выглядеть как-то так:

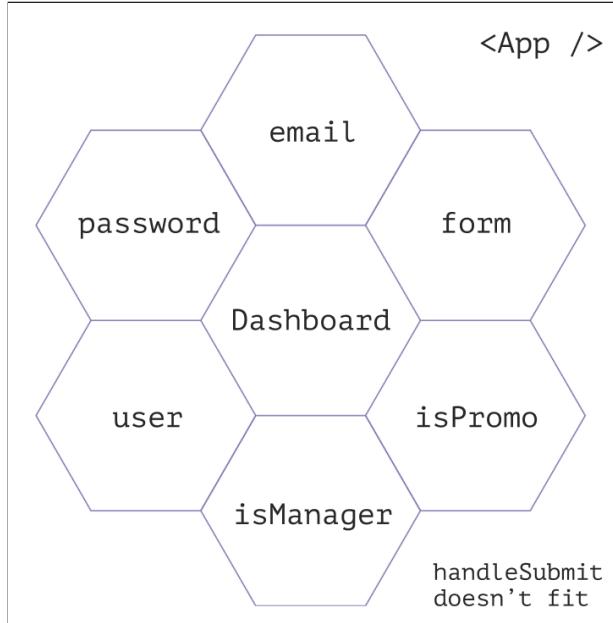
```
const App = () => {
  const user = currentUser();
  const isManager = hasManagerRole(user);
  const isPromoAccount = checkPromoAccount(location);

  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const handleSubmit = () => {
    /*...*/
  };

  return isManager || isPromoAccount ? (
    <Dashboard />
  ) : (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={({ target }) => setEmail(target.value)}
      />
      <input
        type="password"
        password={password}
        onChange={({ target }) => setPassword(target.value)}
      />
      <button>Login</button>
    </form>
  );
};
```

Понять такой код вполне реально, но из-за количества деталей на это уйдёт сравнительно больше времени. Количество информации в этом куске кода подбирается к пределам рабочей памяти нашего мозга.

Если выразить этот код на диаграмме детализации, можно заметить, что некоторые части на ней не помещаются:



Объекты и функции на плитке шестигранников, один из них не помещается

Код будет значительно проще исследовать, если на верхнем уровне мы «подготовим» читателя и расскажем, что делает компонент `<App>`. Имена переменных и подкомпонентов выразят намерение и сложатся в «историю»:

JAVASCRIPT

```
const App = () => {
  const hasAccess = useHasAccess();
  return hasAccess ? <Dashboard /> : <Login />;
};

/**
 * Если пользователям разрешён доступ (`hasAccess`) к панели управления,
 * приложение покажет им компонент панели управления (`Dashboard`),
 * в обратном случае им предложат залогиниться (`Login`).
 */
```

Подробности «истории» мы представим через реализацию соответствующих функций и компонентов. Например, мы расскажем, как определить, есть ли у пользователя доступ к панели управления, через реализацию хука `useHasAccess`:

```

function useHasAccess() {
  const user = currentUser();
  const isManager = hasManagerRole(user);
  const isPromoAccount = checkPromoAccount(location);
  return isManager || isPromoAccount;
}

/**
 * Мы проверим, является ли текущий пользователь ('currentUser')
 * менеджером ('hasManagerRole'),
 * а также запущено ли приложение под промо-аккаунтом,
 * в котором панель управления доступна всем подряд ('checkPromoAccount').
 */

```

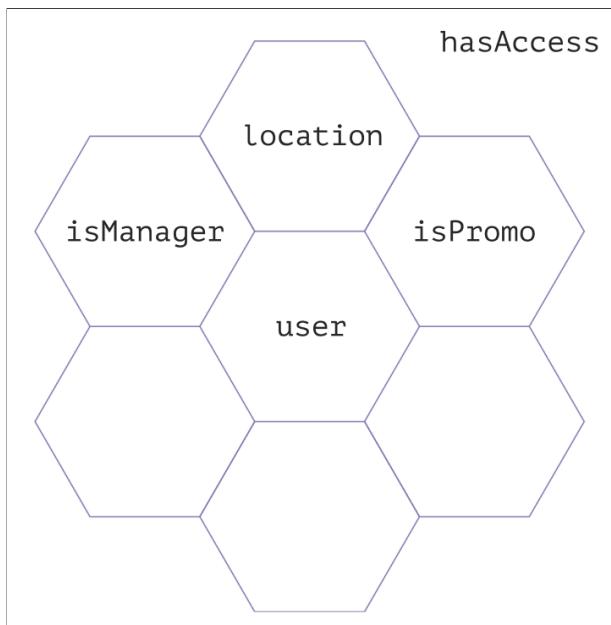
Так на верхнем уровне детализации мы видим всего лишь 3 составные части: `hasAccess`, `Dashboard` и `Login`. Такую схему нам гораздо проще «загрузить» в голову и вникать в отношения между частями.



Верхний слой детализации приложения в виде плитки шестиугранников

Если нам требуется детализировать часть «истории», мы можем «приблизиться» в одну из ячеек и рассмотреть её устройство.

Например, в `useHasAccess` мы видим, как её 4 составные части работают друг с другом. На этом уровне уже не так важно, что происходит «выше», мы сосредотачиваемся на устройстве `useHasAccess`.



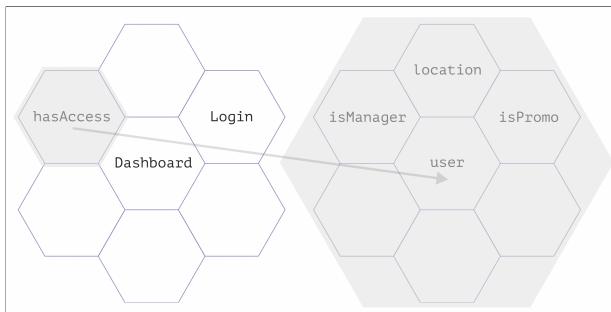
Детализация хука `useHasAccess` в виде плитки

Польза фрактальности в том, что мы можем вкладывать один уровень в другой:



Уровни детализации вкладываются друг в друга как матрёшки

...И переключать внимание между уровнями в любой момент:



Переключение внимания между уровнями

Мы можем приблизить любой шестигранник, чтобы рассмотреть его части. И также мы можем приблизить любую из его частей, погружаясь в детали системы глубже и глубже, но контролируя их количество.

На каждом из уровней нас будет ждать ограниченный *комфортный* объём информации, которую нужно «загрузить» в оперативную память мозга. Исследовать и читать такой код значительно проще.

Абстракция «вшита» в этот подход. Мы можем «отзумиться» на уровень вверх, чтобы посмотреть на взаимодействие модуля с другими либо, наоборот, «провалиться» на уровень вниз, чтобы разглядеть

детали его работы.

К СЛОВУ

Фрактальная архитектура чем-то напоминает концепцию Zoom World из книги “The Humane Interface” Джека Раскина.[♂]

Разделение ответственности

Абстракция заставляет делить код на части, но не всегда очевидно, по каким признакам это делать. Для облегчения этой задачи мы можем использовать принцип *разделения ответственности* (*Separation of Concerns, SoC*) — то есть деления системы на такие части, каждая из которых отвечает только за одну область или задачу.[♂]

«Ответственность», «область» и «задача» — довольно размытые термины. Я привык их понимать, как ограниченный набор данных и процессов, которые связаны *друг с другом сильнее, чем с другими задачами*.

К СЛОВУ

Кто-то мог в этом описании увидеть термин связность.[♂] О ней мы поговорим чуть подробнее в главе об интеграции модулей.

Части хорошо разделённого кода в идеале не перекрывают и не повторяют функциональность друг друга. Работа такой системы строится на результате композиции её частей. При проектировании и рефакторинге такое разделение подталкивает к декомпозиции сложных задач на более простые.

Декомпозиция задач

Когда мы видим большой кусок кода, нам в первую очередь стоит подумать, сколько в нём разных задач. Для этого мы можем посчитать, сколько в этом фрагменте *наборов данных и операций*.

Например, посмотрим на функцию отправки формы логина:

```

async function submitLoginForm(event) {
  const form = event.target;
  const data = {};

  if (!form.email.value || !form.password.value) return;
  data.email = form.email.value;
  data.password = form.password.value;

  const response = await fetch("/api/login", {
    method: "POST",
    body: JSON.stringify(data),
  });
  return response.json();
}

```

Она компактная, всего 13 строк, но в ней можно насчитать 3 задачи:

- извлечение данных из формы;
- валидация данных;
- работа с сетью.

Мы также можем посчитать задачи, изменив условия каждой, и посмотрев, *какой код из-за этого поменяется*. Например, если в форму добавить чекбокс «Запомнить меня», то изменится код извлечения данных:

```

async function submitLoginForm(event) {
  // ...

  data.email = form.email.value;
  data.password = form.password.value;

  // Добавится новое поле в объекте с данными:
  data.rememberMe = form.rememberMe.checked;

  // ...
}

```

А если изменить URL отправки данных, то изменится только часть с API:

```
async function submitLoginForm(event) {  
    // ...  
  
    // Обновится аргумент у `fetch`:  
    const response = await fetch("/api/v2/login", {  
        method: "POST",  
        body: JSON.stringify(data),  
    });  
  
    // ...  
}
```

Такая проверка помогает связывать фрагменты кода и причины для их изменения. Количество разных причин подскажет, сколько на самом деле задач решает код.

К СЛОВУ

Не всегда в большом куске кода много задач. Реализация сложного алгоритма вполне может быть объёмной, но решать всего одну задачу. Это также можно проверить, поменяв условия и посмотрев, какой код меняется.

Принцип единственной ответственности

Код, который меняется по разным причинам, лучше держать отдельно, а который меняется по одинаковой причине — вместе. Это также известно, как *принцип единственной ответственности* (*Single Responsibility Principle, SRP*). ☀ ☀

Мы можем применить этот принцип, чтобы отрефакторить функцию `submitLoginForm` из примера выше. Выделим каждую задачу в отдельную функцию и посмотрим, как изменится код `submitLoginForm`. Начнём с извлечения данных:

```
// Вынесем задачу извлечения данных из формы в отдельную функцию.  
// После такого изменения мы точно знаем, куда «приблизиться»,  
// если нужно узнать детали того, как именно данные извлекаются.  
  
function extractLoginData(form) {  
    const data = {};  
  
    data.email = form.email.value;  
    data.password = form.password.value;  
  
    return data;  
}  
  
  
async function submitLoginForm(event) {  
    const form = event.target;  
  
    // Внутри `submitLoginForm` мы теперь фокусируемся  
    // только на использовании извлечённых данных.  
    const data = extractLoginData(form);  
  
    if (!form.email.value || !form.password.value) return;  
  
    const response = await fetch("/api/login", {  
        method: "POST",  
        body: JSON.stringify(data),  
    });  
    return response.json();  
}
```

Дальше подумаем о валидации. Мы увидим, что валидировать DOM-объект, как раньше, смысла нет. Нам важно проверить данные, но не важно, откуда мы их получаем. Так разделяя ответственность, мы можем избавиться от лишних зависимостей между задачами.

```
// Теперь вся валидация собрана в функции `isValidLogin`.
// Внутри неё проверяем данные, а не свойства DOM-объекта:
function isValidLogin({ email, password }) {
    return !!email && !!password;
}

async function submitLoginForm(event) {
    const form = event.target;
    const data = extractLoginData(form);
    if (!isValidLogin(data)) return;

    const response = await fetch("/api/login", {
        method: "POST",
        body: JSON.stringify(data),
    });
    return response.json();
}
```

Отправку данных и обработку ответа сервера тоже можно выделить в отдельную функцию:

```
// Теперь вся работа с сетью собрана в функции `loginUser`.
async function loginUser(data) {
    const method = "POST";
    const body = JSON.stringify(data);

    const response = await fetch("/api/login", { method, body });
    return await response.json();
}

async function submitLoginForm(event) {
    const form = event.target;
    const data = extractLoginData(form);
    if (!isValidLogin(data)) return;

    return await loginUser(data);
}
```

Получившийся в итоге код проще изменять, потому что выделенные функции ограничивают «зоны ответственности» между задачами. Изменения внутри одной из функций с меньшей вероятностью

будут приводить к изменениям в других. Например, при обновлении правил валидации, поменяется только код функции `isValidLogin`:

JAVASCRIPT

```
function isValidLogin({ email, password }) {  
    // Теперь проверяем, что адрес почты содержит '@':  
    return email.includes "@" && !password;  
  
    // Функции `extractLoginData`, `loginUser` и `submitLoginForm`  
    // останутся нетронутыми.  
}
```

Такое разделение делает тестирование и работу функций в изоляции друг от друга проще. А чем выше изоляция, тем меньше шансов допустить случайную ошибку во время изменения кода.

Инкапсуляция

Принцип единственной ответственности помогает думать о частях кода (функциях, модулях, объектах), как о единицах смысла. То есть как о самостоятельных и осмысленных частях какой-то общей идеи.

Части общаются между собой через API (протокол, контракт, интерфейс) и не лезут во внутренние дела друг друга, чтобы ничего у соседа не сломать. Такое отношение между сущностями мы можем назвать *инкапсуляцией*. 

Инкапсуляцию часто воспринимают как «просто скрытие данных» или ограничение доступа к ним, но...

Главная идея [инкапсуляции] в том, чтобы объект гарантировал, что никогда не будет в невалидном состоянии... [Инкапсулированный] объект сам лучше всех знает, что для него означает «валидность» и как её гарантировать 

Плохая инкапсуляция ведёт к повторяющимся проверкам в коде и ошибкам из-за невалидных состояний данных. Её можно обнаружить по «протёкшим» абстракциям и высокому зацеплению между модулями. О зацеплении мы ещё подробно поговорим в одной из следующих глав, а сейчас остановимся на протёкших абстракциях. Попробуем определить, чем плоха функция

`makePurchase` из примера ниже:

```
// purchase.js
import { createOrder } from "./order";

async function makePurchase(user, cart, coupon) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  order.discount = coupon === "HAPPY_FRIDAY" ? order.total * 0.2 : 0;

  await sendOrder(order);
}
```

Главная проблема этого кода в отсутствии гарантий, что в функцию `sendOrder` попадёт *валидный* заказ. Функция `makePurchase` *меняет* состояние объекта `order`, который был создан *другим* модулем. Она ведёт себя так, будто знает, какое состояние для объекта `order` *валидно*, а какое нет. Но...

Гарантии *валидности* данных — внутренняя задача конкретного модуля

Проще говоря, применить скидку к заказу может тот модуль, который знает, как это сделать *правильно*. В нашем случае это `order.js`: заказ создан в нём, поэтому и применение скидки стоит вынести туда же:

```
// order.js
export function createOrder() {
  /*...*/
}

export function applyDiscount(order, coupon) {
  const discount = coupon === "HAPPY_FRIDAY" ? order.total * 0.2 : 0;
  return { ...order, discount };
}

// purchase.js
import { createOrder, applyDiscount } from "./order";

async function makePurchase(user, cart, coupon) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  const discounted = applyDiscount(order, coupon);
  await sendOrder(discounted);
}

// Теперь гарантии валидности данных заказа —
// это внутренняя задача модуля `order`,
// а не ответственность вызывающего его кода.
```

К СЛОВУ

Во многих языках мы можем запретить менять данные после создания с помощью иммутабельных структур. Тогда привести их в неправильное состояние снаружи не получится.

В JavaScript мы можем делать объекты неизменяемыми с помощью `Object.freeze`, но зачастую это оверхед. Обычно достаточно воспринимать данные, как неизменяемые.

Те же проблемы касаются и проверки корзины на пустоту. И хоть функция `makePurchase` данных корзины не меняет, она всё ещё ведёт себя так, будто знает, какая корзина валидна для покупки, а какая нет.

Проверку на пустоту лучше отдать модулю, который корзину создаёт и знает, какое состояние для неё валидно:

```
// cart.js
export function isEmpty(cart) {
  return !cart.products.length;
}

// purchase.js
import { isEmpty } from "./cart";
import { createOrder, applyDiscount } from "./order";

async function makePurchase(user, cart, coupon) {
  if (isEmpty(cart)) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  const discounted = applyDiscount(order, coupon);
  await sendOrder(discounted);
}
```

В обновлённом коде функция `makePurchase` не меняет состояния заказа напрямую и не решает, валидна ли корзина. Вместо этого она полагается на *публичные API* других модулей.

Это не значит, что после такого изменения автоматически исчезнут все ошибки — в конце концов другие модули могут содержать ошибки и сами. Но по крайней мере мы точно будем знать, где искать проблему, если появится ошибка с данными заказа или валидацией корзины.

Также мы ограничили распространение изменений по кодовой базе в будущем. Пока публичное API модулей не меняется, изменения и правки этих модулей не выйдут за их пределы.

Ну и помимо прочего, такой код проще покрыть тестами и проверять на соответствие проектным требованиям.

1. "Agile Principles, Patterns, and Practices in C#" by Robert C. Martin, <https://www.goodreads.com/quotes/8806618-abstraction-is-the-elimination-of-the-irrelevant-and-the-amplification>
2. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
3. Оценка емкости рабочей памяти, Википедия, https://ru.wikipedia.org/wiki/Рабочая_память#Оценка_емкости_рабочей_памяти
4. "Thinking, Fast and Slow" by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>

5. "Fractal hex flowers" by Mark Seemann, <https://observablehq.com/@ploeh/fractal-hex-flowers>
6. "The Humane Interface" by Jef Raskin, https://www.goodreads.com/book/show/344726.The_Humane_Interface
7. Разделение ответственности, Википедия, https://ru.wikipedia.org/wiki/Разделение_ответственности
8. Связность в программировании, Википедия, [https://ru.wikipedia.org/wiki/%D0%A1%D0%B7%D0%BD%D0%BD%D0%BE%D0%BC%D0%DC%D0%BD%D0%BE%D0%BC_\(%D0%BF%D1%80%D0%BB%D0%BC%D0%DC%D0%BD%D0%BE%D0%BC%D0%BA%D0%BE%D0%BC\)](https://ru.wikipedia.org/wiki/%D0%A1%D0%B7%D0%BD%D0%BD%D0%BE%D0%BC%D0%BD%D0%BE%D1%81%D1%82%D0%BE%D0%BC%D0%BD%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BB%D0%BC%D0%BC%D0%BD%D0%BE%D0%BC%D0%BA%D0%BE%D0%BC))
9. Single Responsibility Principle, Principles of OOD,
<http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
10. Принцип единственности ответственности, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/ru/thing_68/
11. Инкапсуляция в программировании, Википедия, [https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D1%81%D0%BF%D0%DD%D0%BE%D1%83%D1%82%D0%BB%D0%8F_\(%D0%BF%D1%80%D0%BB%D0%BC%D0%DC%D0%DD%D0%BE%D0%BC%D0%BA%D0%BE%D0%BC\)](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D1%81%D0%BF%D0%BD%D0%BE%D1%83%D1%82%D0%BB%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BB%D0%BC%D0%DC%D0%BD%D0%BE%D0%BC%D0%BA%D0%BE%D0%BC))
12. `Object.freeze()`, MDN,
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

Функциональный пайплайн

Как мы увидели в предыдущей главе, абстракция помогает делить программу на «уровни детализации». На каждом из уровней мы уделяем внимание только самым важным деталям и объясняемся подходящими терминами.

В пользовательских приложениях один из уровней абстракции описывает *бизнес-логику* — процессы предметной области, которые делают приложение уникальным и приносят прибыль. Иначе говоря, те задачи, за решением которых бизнес обращается к разработке.

НАПРИМЕР

В случае интернет-магазина бизнес-логикой будет создание и оплата заказов. Для транспортной компании — логистические задачи типа оптимизации маршрутов и загруженности перевозок.

Язык бизнес-логики — это язык предметной области. Процессы в нём описываются как последовательность событий и их последствий: «Когда пользователь вводит купон на скидку, приложение проверит валидность купона и уменьшит сумму заказа».

Такой язык «далёк от кода». Из-за высокого уровня абстракции представить бизнес-процессы в коде может быть сложно. Функциональный пайплайн, о котором мы поговорим в этой главе, помогает описывать бизнес-процессы понятнее и ближе к реальности.

Преобразования данных

Процессы в бизнес-логике — это преобразования данных. Продолжая пример с интернет-магазином, применение скидки к заказу можно выразить в виде перехода от одного состояния данных к другому:

«Применение купона»:

[Созданный заказ] + [Валидный купон] -> [Заказ со скидкой]

В больших приложениях преобразования могут быть длиннее, а данные в них могут проходить через несколько этапов:

«Подбор рекомендаций к корзине»:

[Корзина с товарами] + [История покупок] ->
[Категории товаров] + [Веса рекомендаций] ->
[Список рекомендаций]

В плохо организованном коде бизнес-процессы на такие цепочки не похожи. Они запутаны, неочевидны и часто не говорят на языке предметной области. В итоге вместо последовательного описания бизнес-процесса мы получаем нечто вроде:

«Подбор рекомендаций к корзине»:

[Корзина с товарами] + ... + [Магия 🌟] -> [Список рекомендаций]

В хорошо организованном коде процессы выглядят линейно, а данные в них поочерёдно проходят через несколько этапов. Конечное состояние данных — это желаемый результат всего бизнес-процесса.



Данные проходят по цепочке разных состояний, на выходе получаем желаемый результат

Такая организация кода называется *функциональным пайплайном*. При рефакторинге бизнес-логики мы можем ориентироваться на него, чтобы сделать код понятнее и очевиднее.

Состояния данных

В “Domain Modeling Made Functional” Скотт Влашин рассказывает, как спроектировать программу, опираясь на бизнес-процессы и данные в них.  Он предлагает представлять шаги процессов отдельными функциями. Мы можем использовать эту идею, как основу для рефакторинга.

Для этого в коде нам первым делом потребуется выделить состояния, через которые проходят данные. Эти состояния помогут понять, на какие шаги можно разделить бизнес-процесс, который мы рефакторим.

Разберём подход на примере интернет-магазина. Допустим, функция `makeOrder` составляет объект заказа, учитывая купоны на скидку и промо-акции:

JAVASCRIPT

```
function makeOrder(user, products, coupon) {
    if (!user || !products.length) throw new InvalidOrderDataError();
    const data = {
        createdAt: Date.now(),
        products,
        total: totalPrice(products),
        discount: selectDiscount(data, coupon),
    };

    if (!selectDiscount(data, coupon)) data.discount = 0;
    if (data.total >= 2000 && isPromoParticipant(user)) {
        data.products.push(FREE_PRODUCT_OF_THE_DAY);
    }

    data.id = generateId();
    data.user = user.id;
    return data;
}
```

Функция небольшая, но делает довольно много:

- валидирует пользователя и список товаров;
- создаёт объект заказа;
- применяет скидку по переданному купону;
- добавляет товары по акции, если подходят условия.

С первого взгляда выделить каждый из этих пунктов в коде функции трудно. Код пестрит деталями, которые мешают увидеть отдельные шаги. Из-за этого сложно проследить за изменениями данных и выделить функции для их преобразований.

Объект заказа меняется внутри функции хаотически. И хоть формально его инкапсуляция не нарушена — он создан в этой функции и меняется в ней же — есть ощущение, будто `makeOrder` «лезет в чужую зону ответственности».

Попробуем выделить шаги процесса и состояния, через которые проходят данные:

«Показать заказ на экране»:

- «Проверить входные данные»:

[Непроверенные входные данные] -> [Пользователь] + [Список товаров]

- «Создать заказ»:

[Пользователь] + [Список товаров] -> [Созданный заказ]

- «Применить скидку»:

[Заказ] + [Купон] -> [Заказ со скидкой]

- «Применить акции»:

[Заказ] + [Пользователь] -> [Заказ с акциями]

При рефакторинге будем стремиться к тому, чтобы код функции `makeOrder` стал похож на этот список. Начать можно с группирования кода по «кучкам», каждая из которых будет отвечать за отдельный шаг из него:

```
function makeOrder(user, products, coupon) {  
    // Валидация:  
    if (!user || !products.length) throw new InvalidOrderDataError();  
  
    // Создание заказа:  
    const data = {  
        createdAt: Date.now(),  
        products,  
        total: totalPrice(products),  
    };  
    data.id = generateId();  
    data.user = user.id;  
  
    // Применение скидки:  
    const discount = selectDiscount(data, coupon);  
    data.discount = discount ?? 0;  
  
    // Применение акций:  
    if (data.total >= 2000 && isPromoParticipant(user)) {  
        data.products.push(FREE_PRODUCT_OF_THE_DAY);  
    }  
  
    return data;  
}
```

Группировка шагов поможет найти в коде проблемы с абстракцией: если мы можем придумать для шага осмысленное имя, вероятно, его код можно вынести в функцию. В примере выше это именно так — комментарии с названиями шагов полностью отражают их намерение. Выделим шаги в отдельные функции:

```
// Создание объекта:
function createOrder(user, products) {
  return {
    id: generateId(),
    createdAt: Date.now(),
    user: user.id,

    products,
    total: totalPrice(products),
  };
}

// Применение купона на скидку:
function applyCoupon(order, coupon) {
  const discount = selectDiscount(order, coupon) ?? 0;
  return { ...order, discount };
}

// Применение подходящей акции:
function applyPromo(order, user) {
  if (!isPromoParticipant(user) || order.total < 2000) return order;

  const products = [...order.products, FREE_PRODUCT_OF_THE_DAY];
  return { ...order, products };
}
```

Функция `makeOrder` тогда начала бы выглядеть так:

```
function makeOrder(user, products, coupon) {
  if (!user || !products.length) throw new InvalidOrderDataError();

  const created = createOrder(user, products);
  const withDiscount = applyCoupon(created, coupon);
  const order = applyPromo(withDiscount, user);

  return order;
}
```

После изменений шаги процесса оказались инкапсулированы в отдельных функциях. Единственная задача этих функций — изменить объект заказа, сохранив его валидным. Функция `makeOrder` перестала бесконтрольно менять данные сама и лишь вызывает другие функции. Всё это делает появление невалидного заказа менее вероятным, а тестирование преобразований — проще.

Код функции `makeOrder` теперь напоминает список шагов процесса, с которого мы начинали рефакторинг. Детали каждого из них скрыты за именем соответствующей функции, название которой описывает весь шаг целиком. Это делает код проще для чтения.

Также при добавлении в процесс нового шага нам теперь достаточно вписать его в правильное место. Другие преобразования останутся неизменными:

JAVASCRIPT

```
function makeOrder(user, products, coupon, shipDate) {  
    if (!user || !products.length) throw new InvalidOrderDataError();  
  
    const created = createOrder(user, products);  
    const withDiscount = applyCoupon(created, coupon);  
    const withPromo = applyPromo(withDiscount, user);  
    const order = addShipment(withPromo, shipDate); // Новый шаг процесса.  
  
    return order;  
}
```

А при удалении какого-то шага — проще найти функцию, которую нужно убрать. Убрав вызов функции, мы гарантированно удалим весь код, связанный с этим шагом процесса.

К СЛОВУ

Не всегда пайплайн бывает просто заметить. Преобразования данных могут быть «размазаны» по кодовой базе. В таких случаях может быть полезно составить на бумажке схему общения частей приложения друг с другом, чтобы обнаружить закономерности.

Невыразимость неправильного

Некоторым бизнес-процессам для работы нужны данные только в определённых состояниях. Например, мы не хотим начинать отгрузку заказа, пока он не оплачен или у него не хватает адреса доставки. Такие заказы для этого процесса — невалидны.

Мы можем сделать код надёжнее, если «запретим» передачу невалидных данных. То есть если спроектируем код так, что их передача будет невозможна или значительно сложнее, чем передача валидных данных.

Например, в языках со статической типизацией это можно сделать с помощью типов. Мы можем описать каждое состояние данных отдельным типом и указать, какой тип для какого процесса валиден. Такой код добавит ограничения предметной области прямо в сигнатуру функций и методов.

К СЛОВУ

Понятно, что тут надо делать скидку на особенности и ограничения конкретного языка.

Например, в TypeScript добиться «настоящей валидации в сигнатуре» сложнее, и она будет ограничена JS-рантаймом. Тем не менее даже без неё такая техника помогает отражать больше знаний из предметной области прямо в коде.

Для примера посмотрим на тип `CustomerEmail`, который описывает адрес электронной почты пользователя нашего магазина:

TYPESCRIPT

```
type CustomerEmail = {  
    value: EmailAddress;  
    verified: boolean;  
};
```

У типа есть флаг `verified`, который показывает, верифицирована ли почта. Проблема флага в том, что он не объясняет, при каких условиях будет иметь значение `true`. Из-за этого в типе недостаточно знаний о нюансах верификации почты.

Недостаток этих знаний придётся как-то восполнять, чаще всего — в рантайме с помощью условий

Для примера представим, что в магазине есть ссылка на восстановление аккаунта. При клике она должна отправить пользователя на страницу со сбросом пароля, но только если его почта верифицирована:

TYPESCRIPT

```
function restoreAccount(email: CustomerEmail): void {  
    if (email.verified) {  
        // Отправить пользователя на страницу сброса пароля.  
    } else {  
        return;  
    }  
}
```

Из-за текущей реализации `CustomerEmail` функция `restoreAccount` будет принимать данные, которые в половине случаев для неё невалидны.

Это может быть не страшно, пока тип содержит только один такой флаг. Но чем больше подобных флагов, тем больше разных состояний в этом типе будет находиться одновременно, тем выше вероятность ошибок из-за передачи невалидных данных.

От этого можно перестраховаться, если выделить разные состояния данных в разные типы:

TYPESCRIPT

```
// Неверифицированные адреса будут обозначены одним типом:  
type UnverifiedEmail = {  
    /*...*/  
};  
  
// ...Верифицированные – другим:  
type VerifiedEmail = {  
    /*...*/  
};  
  
// Общий тип обозначим как выбор между первым и вторым:  
type CustomerEmail = UnverifiedEmail | VerifiedEmail;
```

Тогда для разных процессов мы можем требовать разные типы данных:

TYPESCRIPT

```
// Если функции важно, верифицирована почта или нет,  
// она может в сигнатуре указать конкретный тип:  
  
function restorePassword(email: VerifiedEmail): void {}  
function verifyEmail(email: UnverifiedEmail): void {}  
  
// Если функция готова работать с любой почтой, то может использовать общий тип –  
// так сигнатура сама скажет, что верификация для этой функции не важна:  
  
function isValidEmail(email: CustomerEmail): boolean {}
```

Сигнатуры функций стали точнее описывать предметную область, потому что передают больше знаний о предметной области. Функции `restorePassword` и `verifyEmail` предупреждают об ограничениях на входные данные. Функция `isValidEmail` сообщает, что готова работать с любой почтой, и верификация ей не важна.

В случае с TypeScript тип-алиасов, конечно, может оказаться мало. Может потребоваться проследить, чтобы создать неверифицированный адрес почты с типом `VerifiedEmail` было нельзя.

Мы можем использовать для этого брендирование типов \diamond или договориться создавать сущности только с помощью специальных классов или фабрик.

Однако, для описательных целей — передачи знаний о предметной области — алиасов может быть вполне достаточно.

Валидация данных

Функциональный пайплайн опирается на линейное выполнение кода. Шаги внутри процесса выполняются один за другим и передают данные по цепочке друг другу.

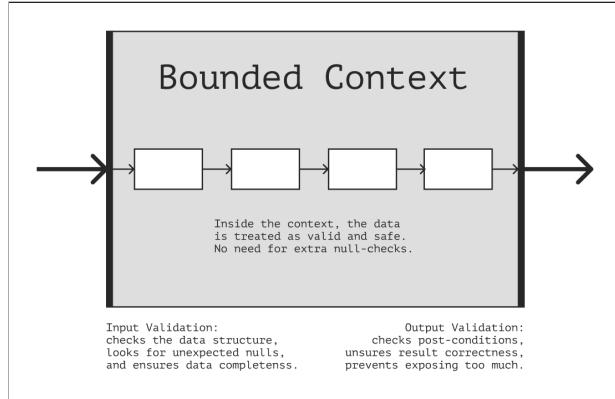
Чтобы эта идея работала, данные внутри процессов должны быть проверенными, безопасными и не ломать пайплайн. Однако, мы не можем гарантировать, что любые данные «извне» будут безопасны. Поэтому в коде мы будем разделять зоны, где данным доверять можно, а где — нет.

ОСТРОВКИ БЕЗОПАСНОСТИ

Бизнес-процесс в идеале должен стать таким «островком», данные внутри которого проверены и безопасны, а снаружи — нет.

В DDD есть аналог таких островков — *ограниченные контексты (Bounded Context)*. $\diamond \diamond \diamond$ Если сильно упростить, то ограниченный контекст — набор функций, которые относятся к некоторой части приложения.

По DDD данные удобнее всего валидировать *на границах* контекстов, например, на входе в контекст перед началом работы. В этом случае «внутри» контекста дополнительные проверки будут не нужны, потому что данные уже будут проверены.



Вся валидация происходит на границах, а внутри данные считаются валидными и безопасными

Мы можем использовать это правило в своём коде, чтобы избавиться от лишних перепроверок во время работы. Проверив данные единожды в начале процесса, мы можем считать, что они соответствуют нашим требованиям.

Тогда, например, в компоненте корзины `CartProducts` вместо ad-hoc проверок на существование товаров и их свойств внутри рендерса:

JAVASCRIPT

```
function CartProducts({ items }) {
  return (
    !!items && (
      <ul>
        {items.map((item) =>
          item ? <li key={item.id}>{item.name ?? "-"}</li> : null
        )}
      </ul>
    );
}
```

...Мы бы один раз проверили данные на входе:

```

function validateCart(cart) {
  if (!exists(cart)) return [];
  if (hasInvalidItem(cart)) return [];

  return cart;
}

// ...

const validCart = validateCart(serverCart);

```

...И внутри компонента уже использовали бы их без лишних проверок:

```

function CartProducts({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      )));
    </ul>
  );
}

```

Пропущенные состояния

Часто валидация на входе помогает нам обнаружать состояния данных, о которых мы не подумали ранее. Например, код компонента `CartProducts` из предыдущего фрагмента стал сильно проще, и поэтому видеть в нём недостатки нам стало легче:

```

// Один из недостатков в том,
// что если корзина валидна, но пуста,
// компонент отрендерит пустой список:

const validEmptyCart = [];
<CartProducts items={validEmptyCart} />

// В разметке получится: <ul></ul>

```

В коде выше состояние «Пустой корзины» валидно, но представляет собой крайний случай. Функциональный пайплайн и предвалидация делают подобные состояния заметнее, потому что те выбиваются из «нормального» выполнения кода. А чем более заметны крайние случаи, тем раньше мы их обнаружим и обработаем:

JAVASCRIPT

```
// Чтобы пофиксить проблему с пустым списком,  
// разделим состояния «Пустой корзины»  
// и «Корзины с товарами» на разные компоненты:  
  
const EmptyCart = () => <p>The cart is empty</p>;  
const CartProducts = ({ items }) => {};  
  
// Тогда при рендере мы сможем сперва обработать все крайние случаи,  
// а уже затем перейти к работе с Happy Path:  
  
function Cart({ serverCart }) {  
  const cart = validateCart(serverCart);  
  
  if (isEmpty(cart)) return <EmptyCart />;  
  return <CartProducts items={cart} />;  
}
```

ОДНАКО

Мы помним, что рефакторинг не должен менять функциональность кода, поэтому исправлять ошибки лучше отдельно. Подробнее о том, как не смешивать рефакторинг с багфиксами, но и не забывать о найденных проблемах в коде, мы поговорим одной и последних глав.

Такой «отсев» крайних случаев, как в компоненте `Cart`, помогает нам обнаружить и учесть больше потенциальных сложностей на ранних этапах разработки. Благодаря этому программа становится надёжнее и точнее описывает предметную область.

К СЛОВУ

Более детально о технике «отсева» крайних случаев и понятии раннего возврата (Early Return) мы поговорим в главе об условиях и сложности кода.

DTO и десериализация

Валидировать данные перед началом работы полезно ещё и потому, что они могут повредиться при сериализации или десериализации. ☺

Как правило, информация между частями системы передаётся в виде *объектов передачи данных* (*Data Transfer Object, DTO*). ↗ ↗ Это такие «пакеты с информацией», которые путешествуют от одной части приложения другой — например, от сервера к клиенту.

Структура и формат DTO намеренно просты: строки, числа, массивы и объекты. Например, у JSON, который часто используют для общения между сервером и клиентом, нет никаких сложных типов или структур.

Во время «перевода» между сложными доменными типами и намеренно простыми DTO что-то может пойти не так, и данные могут оказаться невалидными. Они могут сломать работу процесса, если не проверить их перед использованием.

ПОДРОБНЕЕ

Подробнее о функциональном пайплайне, бизнес-процессах, ограниченных контекстах, валидации данных и DDD писал Скотт Влашин в “Domain Modeling Made Functional”. ↗
Отличная книга, очень рекомендую.

Селекторы и маппинг данных

Одни и те же данные могут быть нужны для различных задач. Например, UI может по-разному показывать на экране список товаров или корзину в зависимости от настроек пользователя.

Функциональный пайpline предлагает «готовить» данные для таких ситуаций заранее. Например, заранее выбирать необходимые фрагменты из исходных данных, трансформировать одни наборы данных в другие или даже сливать несколько наборов в один.

УПРОЩЕНИЕ

По описанию выше вы могли вспомнить какой-то из терминов: маппинг, проекция, слайс, линза, отображение. ↗ ↗ ↗ ↗

Я решил не уделять в этой книге внимание их различиям, чтобы сократить текст и не вводить слишком много новых понятий. Вместо этого я далее по тексту буду использовать слово «выборка» как общий синоним для всех этих терминов, точное значение которого будет зависеть от контекста применения.

Выборки данных помогают «расцепить» модули, которые используют схожие, но слегка отличающиеся данные. Например, посмотрим на компонент `CartProducts`, который рендерит корзину товаров:

```

function CartProducts({ serverCart }) {
  return (
    <ul>
      {serverCart.map((item) => (
        <li key={item.id}>
          {item.product.name}: {item.product.price} × {item.count}
        </li>
      ))}
    </ul>
  );
}

```

Сейчас он полагается на структуру данных корзины, которая приходит с сервера. Если структура изменится — придётся менять и компонент:

```

// Если продукты начнут приходить отдельно,
// то искать конкретный продукт придётся
// прямо во время рендера пункта корзины.

function CartProducts({ serverCart, serverProducts }) {
  return (
    <ul>
      {serverCart.map((item) => {
        const product = serverProducts.find(
          (product) => item.productId === product.id
        ).name;

        return (
          <li key={item.id}>
            {product.name}:{product.price} × {item.count}
          </li>
        );
      ))}
    </ul>
  );
}

```

Выборки данных могут помочь нам «расцепить» ответ сервера и структуру, которую мы используем для рендера. Оформим такую выборку, как отдельную функцию:

```
// Функция `toClientCart` превращает данные от сервера в структуру,
// которой будут пользоваться компоненты приложения.

function toClientCart(cart, products) {
  return cart.map(({ productId, ...item }) => {
    const product = products.find(({ id }) => productId === id);
    return { ...item, product };
  });
}
```

...И будем прогонять данные через эту функцию перед тем, как рендерить их в компоненте:

```
const serverCart = await fetchCart(userId)
const cart = toClientCart(serverCart, serverProducts)

// ...

<CartProducts items={cart} />
```

Компонент тогда будет полагаться на структуру, которую *определяем мы сами*:

```
function CartProducts({ items }) {
  return (
    <ul>
      {items.map(({ id, count, product }) => (
        <li key={id}>
          {product.name} {product.price} × {count}
        </li>
      ))}
    </ul>
  );
}
```

Это значит, что если ответ API поменяется, нам не потребуется обновлять все компоненты, которые используют эти данные. Нам будет достаточно обновить только выборку.

Это особенно полезно, если сервер часто ломает обратную совместимость с клиентским кодом. Этую технику можно считать частным случаем паттерна «Адаптер». ☺

Ещё использовать выборки удобно, если в UI могут быть разные представления одних и тех же данных. Например, кроме самой корзины в магазине может быть список всех доступных товаров с отметками об уже выбранных пользователем.

Для рендера такого списка мы можем переиспользовать уже имеющиеся данные, но слегка иначе их подготовив:

JAVASCRIPT

```
// Используем имеющиеся серверные данные,
// но немного по-другому их готовим:

function toClientShowcase(products, cart) {
  return products.map((product) => ({
    ...product,
    inCart: cart.some(({ productId }) => productId === product.id),
  }));
}
```

Компоненту `Showcase` ничего не нужно знать об ответе сервера. Он будет работать с результатом выборки:

JAVASCRIPT

```
function Showcase({ items }) {
  return (
    <ul>
      {items.map(({ product, inCart }) => {
        <li key={product.id}>
          {product.name} <input type="checkbox" checked={inCart} disabled />
        </li>;
      })}
    </ul>
  );
}
```

Такой подход помогает чётче делить ответственность между кодом: компоненты занимаются только рендером, выборки — подготовкой данных.

Компоненты становятся менее привязанными к остальному коду, потому что полагаются на структуры, которые мы полностью контролируем. Так нам, например, будет проще заменить один компонент другим, если потребуется обновить UI.

Тестируировать изменения данных становится проще, потому что любая выборка — это обычная функция, для её тестирования не нужна сложная инфраструктура. Изменение данных внутри компонента, к примеру, потребовало бы его рендера при тестировании.

У нас появляется больше контроля над данными, которые мы хотим (или не хотим) включить в результат выборки. Мы можем оставлять только те поля, которые использует конкретный компонент и фильтровать всё остальное.

Применять выборки, как правило, удобнее всего сразу после валидации. В этом месте мы уже уверены, что данные безопасны, но *ещё нигде не полагаемся* на их структуру. Это даёт возможность решать, как адаптировать данные под конкретную задачу.

ОБРАТИТЕ ВНИМАНИЕ

Данные в этом примере как бы проходят по цепочке «Непроверенные» → «Валидные» → «Подготовленные» → «Отображённые в UI».

Функциональный пайплайн помогает расписывать любую задачу в виде подобной цепочки. Это облегчает декомпозицию и делает код проще, потому что цепочки помогают выстроить чёткую ментальную модель процесса, который мы выражаем в коде.

-
1. "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
 2. "Branding and Type-Tagging" by Kevin B. Greene, <https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d>
 3. "Bounded Context in DDD" by Martin Fowler, <https://www.martinfowler.com/bliki/BoundedContext.html>
 4. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
 5. Сериализация, Википедия, <https://ru.wikipedia.org/wiki/Сериализация>
 6. Объект передачи данных, DTO, Википедия, <https://ru.wikipedia.org/wiki/DTO>
 7. "Data Mapper" by Martin Fowler, <https://martinfowler.com/eaaCatalog/dataMapper.html>
 8. Projection operations, C# Guide, <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations>
 9. API Slice Overview, Redux Toolkit Docs, <https://redux-toolkit.js.org/rtk-query/api/created-api/overview#api-slice-overview>
 10. "Lenses in Functional Programming" by Albert Steckermeier, <https://sinusoid.es/misc/lager/lenses.pdf>

11. Adapter Pattern, Refactoring Guru, <https://refactoring.guru/design-patterns/adapter>

УСЛОВИЯ И СЛОЖНОСТЬ КОДА

В предыдущей главе мы говорили о пользе функционального пайплайна и линейного выполнения кода. Однако, в реальных приложениях не всегда очевидно, как привести код к такому виду.

Логика сложных приложений часто запутана и содержит много условий. Условия делают код полезным: они описывают поведение программы в разных ситуациях. Но они же делают код сложнее.

В этой главе мы поговорим о том, как распутывать сложные условия, зачем «выпрямлять» код и какие паттерны могут в этом помочь.

Цикломатическая и когнитивная сложность

Чтобы понять, как именно условия делают код сложнее, сперва попробуем определить, что такое «сложность» в принципе.

Одна из эвристик в поиске сложного кода — обратить внимание на его вложенность. Чем она выше, тем код сложнее. Глубоко вложенный код даже не требует вчитываться в него, чтобы оценить его сложность. Мы можем понять что нас ждёт, едва взглянув на фрагмент:

```
function doSomething() {  
    while (...) {  
        if (...) {  
            for (...) {  
                if (...) {  
                    break;  
                }  
                for (...) {  
                    if (...) {  
                        continue  
                    }  
                }  
            }  
        }  
    }  
}
```

Пустое пространство с левой стороны текста показывает, сколько информации содержит этот фрагмент. Чем больше пространство, тем больше деталей придётся держать в голове, тем сложнее код.

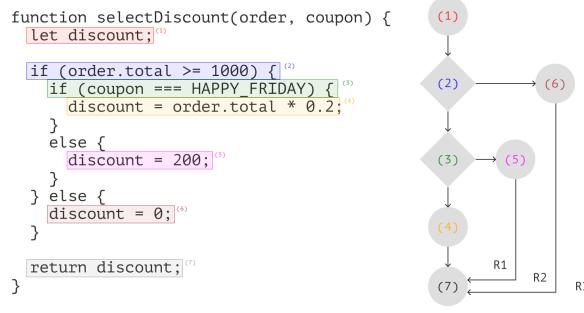
К СЛОВУ

В книге "Your Code as a Crime Scene" Адам Торнхилл называет такое пространство «отрицательным». ☀

При чтении кода мы строим в голове модель его работы. Эта модель описывает, что происходит с данными и в каком порядке выполняются инструкции.

Каждое условие и цикл добавляют в ментальную модель новый способ «пройти» от начала до конца. Чем больше разных «путей», тем сложнее нам одновременно держать их в голове. Количество «путей», по которым можно пройти от начала функции до её конца, называется *цикломатической сложностью* (*Cyclomatic Complexity*) ☀ функции.

Мы можем визуализировать количество «путей», если представим функцию в виде графа. ☀ ☀
Каждое новое условие или цикл добавляет новую «ветку», что делает график функции сложнее.



Граф функции со сложностью 3. Мы можем посчитать сложность как разницу между узлами и рёбрами графа, либо как количество регионов на нём

Чем больше «веток» и сложнее граф, тем труднее нам работать с функцией.

К СЛОВУ

Кроме цикломатической сложности существует ещё *когнитивная* (*Cognitive Complexity*).[♂]

Разница между ними в том, что они считают. Цикломатическая учитывает количество «разных путей выполнения» функции, когнитивная — количество «прерываний линейного выполнения».

Из-за этого говорят, что цикломатическая сложность показывает, насколько тяжело функцию тестировать (сколько веток надо покрыть в тестах), а когнитивная — насколько тяжело её понимать.

Мы не будем заострять внимание на их отличиях, потому что для примеров и техник из этой книги они не будут существенны. Далее по тексту мы будем использовать термин «сложность», как синоним для обеих характеристик.

Важная особенность подобных характеристик в том, что их можно измерить. Для измеряемых характеристик мы можем подобрать лимиты, а их проверку — автоматизировать. Например, мы можем настроить IDE и линтер, чтобы они подсказывали, когда сложность кода превышает выбранный лимит:

function addSymbol(state: KeyboardState, value: string): KeyboardState
Function 'addSymbol' has a complexity of 12. Maximum allowed is 10. eslint(complexity)
View Problem Quick Fix... (#.)

```

function addSymbol(state: KeyboardState, value: string): KeyboardState

```

Линтер ругается на код с цикломатической сложностью, превышающей лимит

Конкретное число зависит от характеристики, проекта, языка и команды. Для цикломатической сложности Марк Симанн в "Code That Fits in Your Head" предлагает использовать число 7. Википедия предлагает ориентироваться на число 10. ↩ ↩ Я в своих проектах обычно использую число 10, потому что оно «круглое», но это не принципиально.

Плоское лучше вложенного

Отрицательное пространство и большая вложенность — следствие сложности кода. Чтобы сделать код проще, мы при рефакторинге можем придерживаться эвристики:

Сделать условия плоскими. ...А потом подумать, как ещё упростить код

Плоские условия «выпрямляют» исполнение кода. Они избавляют нас от необходимости одновременно учитывать *много разных вариантов* работы функции и помогают увидеть закономерности в коде, которые могут упростить его ещё сильнее. Чем плосче код, тем проще увидеть общую картину и суть работы функции или модуля.

К СЛОВУ

Наши цели совпадают с одним из принципов дзена Python: «Плоское лучше вложенного». ↩
На мой взгляд, это подтверждает, что мы на правильном пути.

Ранний возврат

Наиболее простая и частая техника, которую мы можем применить для «уплощения» кода, — это ранний возврат. Он помогает «отсеять» ненужные условия в начале функции и больше не держать их в голове.

Для примера посмотрим на функцию `usersController`, которая создаёт нового пользователя по переданному имени и адресу почты. Условия в этой функции заставляют держать в голове *все свои ветки* до самого конца, потому что полезные действия находятся в каждой из них:

```

function usersController(req, res) {
  if (req.body) {
    const { name, email } = req.body;
    if (name && email) {
      const user = createUser({ name, email });
      if (user) {
        res.json({ user });
      } else {
        res.status(500);
      }
    } else {
      res.status(400).json({ error: "Name and email required" });
    }
  } else {
    res.status(400).json({ error: "Invalid request payload." });
  }
}

```

Можно заметить, однако, что код внутри блоков `else` обрабатывает «крайние случаи» — разные ошибки и невалидные данные. Мы можем упростить функцию, «вывернув» условие и сперва обработав их, оставив на конец лишь работу с “happy path”:

```

function usersController(req, res) {
  if (!req.body) {
    return res.status(400).json({ error: "Invalid request payload." });
  }

  const { name, email } = req.body;
  if (!name || !email) {
    return res.status(400).json({ error: "Name and email required" });
  }

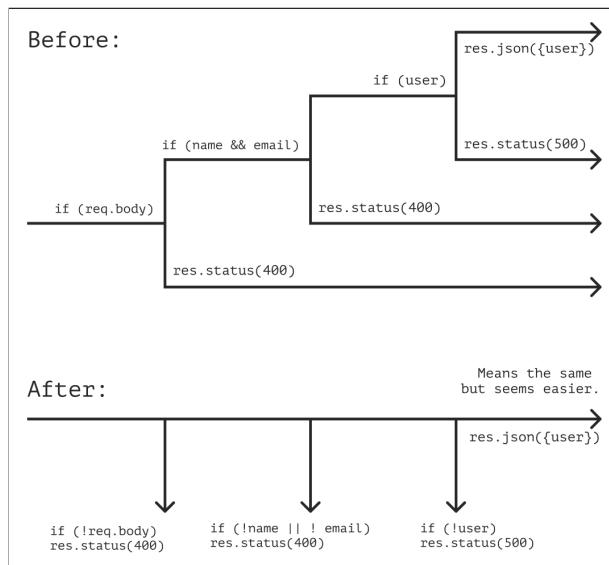
  const user = createUser({ name, email });
  if (!user) return res.status(500);

  res.json({ user });
}

```

Общее количество информации и ситуаций, которые функция обрабатывает, не изменилось. Но мы уменьшили количество веток, за которыми надо следить одновременно.

Мы по очереди проверяем крайние случаи и завершаем работу функции, если наткнулись на один из них. Если функция продолжает работу, значит этих ситуаций не случилось. Так мы будто «отсеиваем» условия по одному за раз, поэтому проверенные ветки не отнимают внимание и рабочую память.



Условие становится проще воспринимать, потому что нам не нужно держать в памяти много деталей

ОДНАКО

Ранний возврат может не подойти для «защитного» программирования: когда мы намеренно и подробно описываем каждую ветку условия.

В моём опыте проектов, где требовалось защитное программирование, было не очень много. По умолчанию я использую ранний возврат, а к защитному программированию перехожу в исключительных случаях.

Рендер компонентов

Ранний возврат может быть полезен для упрощения рендеринга React-компонентов. Например, код компонента `Soveg` из-за взаимозависимых условий прочесть сложно:

```
function Cover({ error, isLoading, data }) {
  if (!error && !isLoading) {
    const image = data.image ?? DEFAULT_COVER_IMAGE;
    return <Picture src={image} />;
  } else if (isLoading) {
    return <Loader />;
  } else {
    return <Error message={error} />;
  }
}
```

Мы можем упростить его, «вывернув» условие и сперва обработав состояния загрузки и ошибки:

```
function Cover({ error, isLoading, data }) {
  if (isLoading) return <Loader />;
  if (error) return <Error message={error} />;

  const image = data.image ?? DEFAULT_COVER_IMAGE;
  return <Picture src={image} />;
}
```

Мы помним, что *количество веток* в условии после применения раннего возврата остаётся таким же. Это значит, что сложность кода может всё ещё оставаться высокой. Нам стоит посмотреть на метрики и узнать, стал ли код проще.

Если сложность функции всё ещё высокая, то стоит проверить, нет ли в ней проблем с абстракцией или разделением ответственности. Вероятно, мы заметим, что функцию можно разбить на несколько:

```
// Например, мы можем разделить «рендер картинки»
// и «решение, какой компонент показывать».
//
// Для этого вынесем рендер картинки в `CoverPicture`
// и будем использовать его, когда данные загрузились,
// а ошибок точно нет.

function CoverPicture(image) {
  const source = image ?? DEFAULT_COVER_IMAGE;
  return <Picture src={source} />;
}

// Решение о том, какой компонент показать,
// останется в компоненте `Cover`.
// Он будет определять, что показывать:
// - `Loader`;
// - `Error`;
// - или `CoverPicture`.

function Cover({ error, isLoading, data }) {
  if (isLoading) return <Loader />;
  if (error) return <Error message={error} />;
  return <CoverPicture image={data.image} />;
}
```

После подобного рефакторинга нам стоит обратить внимание на имена изменённых функций и компонентов. Старое имя может оказаться неточным после того, как мы чётче разделили ответственность между сущностями. В примере выше нам стоит подумать, насколько адекватно имена `Cover` и `CoverPicture` отражают суть этих компонентов.

К СЛОВУ

В целом, для работы с UI больше подходят конечные автоматы, \odot чем ранний возврат. Но если в нашем проекте их нет, а внедрить их по каким-либо причинам нельзя, то ранний возврат может сильно упростить код.

Идейно ранний возврат в рендере похож на валидацию данных перед началом бизнес-процесса. Просто здесь вместо невалидных данных мы отсеиваем «неправильные» состояния UI.

Переменные, предикаты и булева алгебра

Не каждое условие получится «вывернуть». Если ветки условия тесно переплетены, мы можем не найти, откуда начать его распутывать. Чтобы упростить такие условия, в них надо найти закономерности и паттерны.

Первым делом нам стоит вынести повторяющиеся части условий в переменные. Названия переменных помогут абстрагировать детали и сосредоточиться на смысле условия. Это поможет увидеть, можно ли упростить условие с помощью логических правил.

Законы де Моргана

Законы де Моргана — это набор правил, связывающих пары логических операций через отрицание.⁸ Они помогают «разворачивать» скобки в условиях:

```
!(A && B) === !A || !B;  
!(A || B) === !A && !B;
```

Мы можем использовать законы де Моргана, что сделать условие менее шумным. Например, в условии ниже слишком много деталей, «за буквами» сложно увидеть его смысл:

JAVASCRIPT

```
if (user.score >= 50) {  
    if (user.armor < 50 || user.resistance !== 1) {  
    }  
} else if (user.score < 50) {  
    if (user.resistance === 1 && user.armor >= 50) {  
    }  
}
```

В качестве первого шага мы можем вынести повторяющиеся части выражений в переменные:

JAVASCRIPT

```
const hasHighScore = user.score >= 50;  
const hasHeavyArmor = user.armor >= 50;  
const hasResistance = user.resistance === 1;
```

...Тогда условие превратится в сочетания этих переменных:

```

if (hasHighScore) {
  if (!hasHeavyArmor || !hasResistance) {
  }
} else if (!hasHighScore) {
  if (hasResistance && hasHeavyArmor) {
  }
}

```

В таком условии гораздо проще заметить паттерны внутри блоков `if` и упростить их. В частности, мы можем применить первый закон де Моргана, чтобы упростить сочетания переменных `hasHeavyArmor` и `hasResistance`:

```

// Вынесем 2-е вложенное условие в переменную:
const hasAdvantage = hasHeavyArmor && hasResistance;

// Заметим, что по первому закону де Моргана
// 1-е вложенное условие превратится в `!hasAdvantage`:
//
// !A || !B === !(A && B)
//
// A -> hasHeavyArmor
// B -> hasResistance
//
// !hasHeavyArmor || !hasResistance
//     === !(hasHeavyArmor && hasResistance)
//     === !hasAdvantage

// Тогда всё условие станет выглядеть так:
if (hasHighScore && !hasAdvantage) {
} else if (!hasHighScore && hasAdvantage) {
}

```

Предикаты

Не все условия можно вынести в переменную. Например, это сделать сложнее, если условие само зависит от других переменных или изменяющихся данных.

Для таких случаев мы можем использовать *предикаты* — функции, которые принимают произвольное количество аргументов и возвращают `true` или `false`. Функция `isAdult` из фрагмента ниже — пример предиката:

```
const isAdult = (user) => user.age >= 21;

isAdult({ age: 25 }); // true
isAdult({ age: 15 }); // false
```

В предикатах можно хранить «схему» условия. То есть описать принцип, по которому мы будем проводить сравнение разных значений:

```
// Условие в примере ниже сравнивает разные переменные
// с одинаковым эталонным значением – 21:
if (user1.age >= 21) {
} else if (user2.age < 21) {
}

// Мы можем вынести «схему» этого сравнения в предикат `isAdult`
// и использовать его с разными переменными:
if (isAdult(user1)) {
} else if (!isAdult(user2)) {
}
```

Названия предикатов описывают смысл проверки, которую проводит условие. Это помогает чинить «протекающие» абстракции и делать код менее шумным. Например, во фрагменте ниже сложно сходу понять, что проверяет условие:

```
if (user.account < totalPrice(cart.products) - cart.discount) {
    throw new Error("Not enough money.");
}
```

Если мы вынесем сравнение в предикат `hasEnoughMoney`, то по названию функции нам будет проще понять его смысл:

```

function hasEnoughMoney(user, cart) {
    return user.account >= totalPrice(cart.products) - cart.discount;
}

if (!hasEnoughMoney(user, cart)) {
    throw new Error("Not enough money.");
}

```

Кроме этого предикаты, как и переменные, помогают замечать закономерности в сложных условиях и упрощать их с помощью булевой логики.

ПРИМИТИВНЫЙ ПАТТЕРН-МАТЧИНГ

Отдельно при рефакторинге нам стоит обращать внимание на множественные повторения

`else-if`. Если такие условия занимаются выбором какого-то значения, их можно заменить более декларативными и безопасными способами.

Например, посмотрим на функцию `showErrorMessage`, которая сопоставляет тип ошибки валидации и сообщение для неё:

```

type ValidationMessage = string;
type ValidationError = "MissingEmail" | "MissingPassword" | "TooShortPassword";

function showErrorMessage(errorType: ValidationError): ValidationMessage {
    let message = "";

    if (errorType === "MissingEmail") {
        message = "Email is required.";
    } else if (errorType === "MissingPassword") {
        message = "Password is required.";
    }

    return message;
}

```

По задумке функция должна проверить все возможные варианты `ValidationMessage` и выбрать сообщение. В функции, однако, пропущен вариант ошибки для `TooShortPassword`, а компилятор

TypeScript этого не заметил. Без специальных проверок (Exhaustiveness Check \wp) во множественных условиях легко пропустить вариант и не заметить этого.

Множественные условия можно заменить на более декларативные конструкции. Например, мы можем собрать все сообщения об ошибках в словарь, где ключами будут типы ошибок, а значениями — сообщения. Выбор ошибки по такому словарю может оказаться надёжнее, потому что компилятор будет следить за отсутствующими ключами:

TYPESCRIPT

```
function showErrorMessage(errorType: ValidationMessage): ValidationMessage {
    // В типе переменной `messages` явно укажем,
    // что в ней должны быть перечислены все возможные ошибки:
    const messages: Record<ValidationMessage, ValidationMessage> = {
        MissingEmail: "Email is required.",
        MissingPassword: "Password is required.",

        // Если какого-то ключа будет не хватать, компилятор об этом скажет:
        // "Property 'TooShortPassword' is missing in type..."
    };

    return messages[errorType];
}
```

Идейно это похоже на паттерн-матчинг. \wp Мы сопоставляем `errorType` с ключами объекта `messages` и выбираем по нему подходящее значение.

Такой выбор чем-то похож на работу `switch`, только проверка типов в этом случае не требует дополнительных действий (например, Exhaustiveness Check). В TypeScript это, пожалуй, самый дешёвый способ имитировать типобезопасный «паттерн-матчинг» без использования сторонних библиотек.

К СЛОВУ

Стоит отметить, что эта техника больше подходит для простых сопоставлений и выглядит не так красиво, как настоящий паттерн-матчинг в функциональных языках. \wp

Нативный синтаксис для паттерн-матчинга в JS пока находится в Stage 1. \wp Для более сложных сопоставлений можно использовать сторонние библиотеки типа `ts-pattern`.
 $\wp \wp$

В коде на JavaScript подобной проверки типов нет, но декларативность подхода делает и ручной поиск ошибок проще. Чем более явно мы сопоставляем ключ и значение, тем проще заметить несоответствия.

О декларативности мы детально поговорим в одной из следующих глав.

Стратегия

Множественные `else-if` условия могут выбирать не просто значение, а дальнейшее поведение программы. Если условие выбирает поведение среди однотипных вариантов, это может говорить о смешении ответственности или недостаточном полиморфизме. ☺

Один из вариантов решения этой проблемы похож на техники рефакторинга из предыдущего раздела.

Для примера посмотрим на функцию `notifyUser`. Она показывает пользователю сообщение одним из трёх возможных способов. Выбор конкретного способа зависит от условий внутри этой функции:

TYPESCRIPT

```
function notifyUser(message) {
    if (method === methods.popup) {
        const popup = document.querySelector(".popup");
        popup.innerText = message;
        popup.style.display = "block";
    } else if (method === methods.remote) {
        notificationService.setup(TOKEN);
        notificationService.sendMessage(message);
    } else if (method === methods.hidden) {
        console.log(message);
    }
}
```

Проблема функции в том, что она смешивает разные задачи. Она занимается *и выбором способа отправки сообщения, и самой отправкой*. Это значит, что добавление нового способа или изменение существующего затронет всю функцию целиком. Причём даже те части, которые к изменениями отношения не имеют.

Из-за этого использовать отдельно один из способов отправки сообщений в другом коде не получится. Протестировать разные способы изолированно друг от друга тоже не выйдет. А при добавлении новых способов, функция `notifyUser` станет заметно объёмнее и сложнее.

Вместо этого мы можем разделить выбор действия и непосредственно *действие*:

```
// Выделим каждый способ отправки сообщений в отдельную функцию:  
function showPopupMessage(message) {  
    const popup = document.querySelector(".popup");  
    popup.innerText = message;  
    popup.style.display = "block";  
}  
  
// Если сигнатура функции отправки не совпадает с остальными:  
function notifyUser(token, message) {  
    notificationService.setup(token);  
    notificationService.sendMessage(message);  
}  
  
// ...Мы можем её адаптировать:  
const showNotificationMessage = (message) => notifyUser(TOKEN, message);  
  
// Далее подготовим набор функций отправки,  
// по одной на каждый вариант из `methods`:  
const notifiers = {  
    [methods.popup]: showPopupMessage,  
    [methods.hidden]: console.log,  
    [methods.remote]: showNotificationMessage,  
};  
  
// При использовании выбираем поведение  
// в зависимости от параметра `method`:  
function notifyUser(message) {  
    const notifier = notifiers[method] ?? defaultNotifier;  
    notifier(message);  
}  
  
// Выбрать функцию можно и заранее,  
// если `method` известен до исполнения функции.
```

Такая реализация упростит добавление и удаление вариантов отправки сообщений:

```
// Чтобы добавить новый способ,
// достаточно добавить новую функцию...
function showAlertMessage(message) {
    window.alert(message);
}

const notifiers = {
    // ...
    // ...И указать её, как вариант выбора:
    [methods.browser]: showAlertMessage,
};

// Остальной код останется неизменным.
```

Изменения отдельно взятой функции не выйдут за её пределы и не повлияют на `notifyUser` или другие варианты отправки сообщений. Тестировать и использовать такие функции независимо друг от друга гораздо проще.

Разделение выбора и действия — это по сути паттерн «Стратегия». [♂] Его может быть трудно увидеть в коде без классов, потому что примеры этого паттерна чаще всего показывают в парадигме ООП, но это он. Просто если в ООП каждый вариант поведения — это класс, то в нашем примере — это функции.

Null-объект

Лишние условия и проверки также могут появляться из-за однотипной, но слегка отличающейся функциональности в приложении.

Для примера представим, что мы пишем мобильное приложение под iOS, Android и веб. В версии для мобильных платформ мы хотим добавить виджеты. У iOS и Android есть нативные API для обновления виджетов, для которых у нас есть JS-адаптеры.

Функциональность адаптеров описана в интерфейсе `Device`. Они содержат идентификатор платформы и метод для обновления виджета:

```
interface Device {
    platform: Platform;
    updateWidget(data: WidgetData);
}
```

В вебе виджетов нет, поэтому их функциональность для веба мы решаем отключить. Один из путей это сделать — определять платформу и включать фичу по условию:

```
const iosDevice: Device = {
    platform: Platform.Ios,
    updateWidget: (data) => {}, // Логика с нативными iOS API...
};

const androidDevice: Device = {
    platform: Platform.Android,
    updateWidget: (data) => {}, // Логика с нативными Android API...
};

function update(device: Device, data: WidgetData) {
    if (
        device.platform === Platform.Android ||
        device.platform === Platform.Ios
    ) {
        // Вызываем `updateWidget` только для iOS и Android:
        device.updateWidget(data);
    }
}
```

Пока таких проверок одна или две, они нам вряд ли помешают. Но чем больше фич мы будем так проверять, тем больше условий в коде появится. Если мы видим, что проверки приходится делать часто, мы можем воспользоваться вторым вариантом решения проблемы.

Добавим объект-«пустышку» для веба, который будет реализовывать интерфейс `Device`, но в ответ на вызов `updateWidget` ничего не будет делать:

```
const webDevice: Device = {
    platform: Platform.Web,
    updateWidget() {},
};
```

Такие «пустышки» называются *null-объектами*.[♂] Как правило, их добавляют в места, где нужен вызов метода, но не нужна реализация этого вызова.

При использовании null-объекта нам больше не нужно проверять тип устройства — достаточно просто вызвать метод. Если текущий девайс — это веб-браузер, то вызов метода ни к чему не приведёт:

```
function update(device: Device, data: WidgetData) {
    device.updateWidget(data);

    // webDevice.updateWidget();
    // void;

    // Условие больше не нужно.
}
```

Null-объект иногда считают **антипаттерном**.[♂] Его уместность может зависеть от задачи, целей использования и предпочтений команды. Перед применением лучше посоветоваться с другими разработчиками и убедиться, что ни у кого нет возражений против использования этого паттерна.

1. "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>
2. Цикломатическая сложность, Википедия, https://ru.wikipedia.org/wiki/Цикломатическая_сложность
3. Граф потока управления, Википедия, https://ru.wikipedia.org/wiki/Граф_потока_управления
4. Control flow graph & cyclomatic complexity for following procedure, Stackoverflow, <https://stackoverflow.com/a/2670135/3141337>
5. "Cognitive Complexity. A new way of measuring understandability" by G. Ann Campbell, SonarSource SA, <https://www.sonarsource.com/docs/CognitiveComplexity.pdf>
6. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>

7. The Zen of Python, <https://peps.python.org/pep-0020/#the-zen-of-python>
8. Управление состоянием приложения с помощью конечного автомата, <https://bespoyasov.ru/blog/fsm-to-the-rescue/>
9. Законы де Моргана, Википедия, https://ru.wikipedia.org/wiki/Законы_де_Моргана
10. `switch-exhaustiveness-check`, ES Lint TypeScript, <https://typescript-eslint.io/rules/switch-exhaustiveness-check/>
11. Сопоставление с образцом, Википедия, https://ru.wikipedia.org/wiki/Сопоставление_с_образцом
12. ECMAScript Pattern Matching Proposal, <https://github.com/tc39/proposal-pattern-matching>
13. `ts-pattern`, Library for Pattern Matching in TypeScript, <https://github.com/gvergnaud/ts-pattern>
14. "Bringing Pattern Matching to TypeScript" by Gabriel Vergnaud, <https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k>
15. Pattern matching in Haskell, Learn You Haskell, <http://learnyouahaskell.com/syntax-in-functions#pattern-matching>
16. «Полиморфизм простыми словами» Sergey Ufocoder, <https://medium.com/devschacht/polymorphism-207d9f9cd78>
17. Strategy Pattern, Refactoring Guru, <https://refactoring.guru/design-patterns/strategy>
18. Introduce Null Object, Refactoring Guru, <https://refactoring.guru/introduce-null-object>
19. Null object pattern, Criticism, Wikipedia, https://en.wikipedia.org/wiki/Null_object_pattern#Criticism

Сайд-эффекты

Любое взаимодействие программы с внешним миром — это сайд-эффект. Сохранение данных на сервере, вывод на экран, доступ к браузерному API — всё это эффекты. Они необходимы, чтобы написать полезное приложение, но работать с ними сложно и чревато ошибками.

В этой главе обсудим, как проектировать программы и рефакторить код, чтобы эффекты не мешали читать и понимать код. Разберём, в чём польза функционального программирования и неизменяемых структур данных. Рассмотрим «бутерброд» из эффектов и разницу между CQS и CQRS.

Чистые функции

Главная проблема эффектов в их *непредсказуемости*. Они меняют состояние вокруг себя, поэтому мы не можем быть уверены, что результат работы кода будет всегда одинаков.

Чистые функции наоборот эффектов не производят, от состояния не зависят и всегда возвращают одинаковый результат при одинаковых аргументах. Это делает результат их работы предсказуемым и воспроизводимым.

При рефакторинге мы будем стараться чаще использовать чистые функции и выражать больше функциональности через них. Но чтобы понять зачем, сперва обсудим пользу чистых функций.

Ссылочная прозрачность

Проще всего преимущество чистых функций увидеть во время отладки. Для быстрого поиска багов в проблемном участке кода, удобно использовать двоичный поиск. То есть поделить фрагмент кода пополам, определить в какой из половин ошибки и дальше искать баг только в этой половине:

Укажем схематично проблемный кусок кода,
ошибку в котором отметим как «Х»:

[.....X.....]

1.

Чтобы найти ошибку в этом куске,
поделим функциональность пополам
и проверим каждую половину:

[.....|...X.....]

2.

Если левая половина работает нормально,
значит – ошибка в правой:

[.....X.....]

3.

Делим правую половину пополам
и проверяем каждую часть в ней:

[.....X..|.....]

4.

Правая часть работает нормально,
значит – ошибка в левой:

[.....X..|.....]

5.

Снова делим проблемный участок пополам,
проверяя по очереди каждую часть
и так далее, пока не найдём точное место с ошибкой:

[...|X..|.....]

[.....|X..|.....]

[.....|X|.|.....]

[.....|X|.....]

С каждой итерацией проблемный фрагмент кода
уменьшается в два раза – это сильно экономит время.

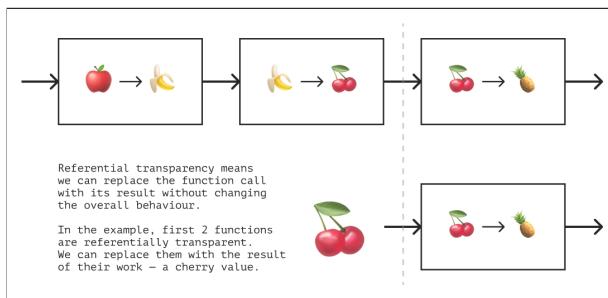
Как правило, через 3–6 итераций становится понятно, в чём именно проблема и где она находится.

К СЛОВУ

Иногда такой процесс двоичного поиска ещё называют бисекцией по аналогии с бисекцией в гите. ☺

Польза чистых функций в том, что их последовательность мы так можем «разрезать» в любом месте, потому что они не связаны друг с другом общим состоянием. Функциям, стоящим после «разреза», не важно, сколько функций и каких было вызвано до них. Если им на вход передают одинаковые аргументы, они будут работать одинаково.

То есть мы буквально можем заменить вызовы функций до «разреза» на результаты их работы, и это не изменит поведения программы. Такое свойство называется *ссылочной прозрачностью* (*Referential Transparency*), ☺ и оно делает поиск ошибок и тестирование намного проще.



Ссылочная прозрачность позволяет заменить вызов функции результатом её работы

С кодом, построенным на сайд-эффектах, такое провернуть гораздо сложнее. Поэтому при рефакторинге мы будем уменьшать количество эффектов и выражать большую функциональности через чистые функции.

Неизменяемость по умолчанию

Как мы говорили ранее, код с эффектами менее предсказуем. Во время его рефакторинга нам в первую очередь стоит проверить, можно ли переписать его без использования эффектов. На практике это чаще всего означает замену общего состояния, связывающего функциональность, на цепочки преобразований данных.

Для примера посмотрим на функцию `prepareExport`, которая готовит данные заказа магазина к экспорту. Она считает подытог по каждой позиции и ищет самую позднюю дату отправки для каждой позиции.

JAVASCRIPT

```
function prepareExport(items) {
    let latestShipmentDate = 0;

    for (const item of items) {
        item.subtotal = item.price * item.count;

        if (item.shipmentDate >= latestShipmentDate) {
            latestShipmentDate = item.shipmentDate;
        }
    }

    for (const item of items) {
        item.shipmentDate = latestShipmentDate;
    }

    return items;
}
```

Действия внутри функции меняют объекты в массиве `items`. При этом соседние действия также зависят от этого массива, и изменения внутри него повлияют на их работу тоже.

Это значит, например, что определяя дату отправки, нам придётся думать, как изменения в `items` отразятся на подсчёте итоговой стоимости. Чем объёмнее функция, тем больше действий затронет эффект, тем больше деталей придётся держать в голове.

Более того, так как `items` — массив, его изменения будут видны и *снаружи* функции `prepareExport`. Эффект будет влиять на код вне функции, о котором мы можем ничего не знать. В этом случае предусмотреть все потенциальные проблемы будет вовсе невозможно.

Вместо попыток уследить за влиянием эффектов, мы можем попробовать их избежать. Перепишем код так, чтобы не менять массив `items`, а представить задачу как набор отдельных последовательных шагов.

Результатом каждого шага будет *новый* набор данных, а сами шаги не будут влиять друг на друга через общее состояние:

```
function prepareExport(items) {  
    // 1. Посчитаем подытог.  
    //     Результат представим как новый массив.  
    const withSubtotals = items.map((item) => ({  
        ...item,  
        subtotal: item.price * item.count,  
    }));  
  
    // 2. Посчитаем дату отправки.  
    //     В качестве исходных данных будем  
    //     использовать результат прошлого шага.  
    let latestShipmentDate = 0;  
    for (const item of withSubtotals) {  
        if (item.shipmentDate >= latestShipmentDate) {  
            latestShipmentDate = item.shipmentDate;  
        }  
    }  
  
    // 3. Проставим дату в каждую позицию.  
    //     Результат снова представим  
    //     в виде нового массива.  
    const withShipment = items.map((item) => ({  
        ...item,  
        shipmentDate: latestShipmentDate,  
    }));  
  
    // 4. Вернём результат шага 3,  
    //     в качестве результата всей задачи.  
    return withShipment;  
}
```

Шаги мы можем вынести в отдельные функции и, если требуется, отрефакторить каждую отдельно:

```

function calculateSubtotals(items) {
  return items.map((item) => ({ ...item, subtotal: item.price * item.count }));
}

function calculateLatestShipment(items) {
  const latestDate = Math.max(...items.map((item) => item.shipmentDate));
  return items.map((item) => ({ ...item, shipmentDate: latestDate }));
}

```

Тогда начальная функция `prepareExport` будет выглядеть как результат последовательности из преобразований данных:

```

function prepareExport(items) {
  const withSubtotals = calculateSubtotals(items);
  const withShipment = calculateLatestShipment(withSubtotals);
  return withShipment;
}

// items -> withSubtotals -> withShipment

```

Или даже так, если мы используем Hack Pipe Operator, который на момент написания находится в Stage 2: ⚡

```

const prepareExport =
  items |> calculateSubtotals()% |> calculateLatestShipment(%);

```

Выделять шаги в отдельные функции нужно не всегда, но это удобно в нескольких случаях:

- Если нам нужно протестировать каждый шаг отдельно от других.
- Если мы хотим использовать преобразования в других местах приложения.
- Если мы хотим сделать каждое состояние, через которое проходят данные, более явным.

Заметим, что в новой реализации функция `prepareExport` *не меняет* исходные данные. Вместо этого она создаёт копию данных и меняет её. Массив `items` остаётся неизменным, что предотвращает ошибки в коде снаружи функции.

Шаги внутри `prepareExport` теперь связаны друг с другом только через свои входные и выходные данные. У них больше нет общего состояния, которое могло бы повлиять на их работу. Благодаря этому нам проще строить в голове модель работы всей функции `prepareExport` в целом.

Абстрагируя каждый шаг в отдельную функцию с понятным именем, мы делаем смысл всей цепочки более явным. Это помогает концентрироваться на отдельных шагах, не отвлекаясь на соседние. Изоляция же помогает на каждом шаге обеспечить валидность данных, потому что не даёт повлиять на работу функции «снаружи».

Создание копий данных на каждый новый шаг функции может быть требовательным к памяти и производительности. Во фронтенде это обычно проблем не вызывает, но всё же стоит иметь это в виду. Если мы гонимся за производительностью, мутабельный подход может подойти лучше.

К СЛОВУ

В JavaScript с настоящей неизменяемостью тяжело. Для действительно неизменяемых объектов понадобится `Object.freeze`, который используют довольно редко.

Но «настоящая» неизменяемость не всегда нужна. Чаще всего достаточно *писать и воспринимать* код так, будто он работает с неизменными данными.

Функциональное ядро в императивной оболочке

Неизменяемость и чистые преобразования — это хорошо, но, как мы упоминали в самом начале, совсем без эффектов обойтись нельзя. Взаимодействие с внешним миром — получение и сохранение данных или вывод их в UI — это всегда эффекты. Без такого взаимодействия приложение будет бесполезным.

Так как проблема эффектов в их непредсказуемости, то наша главная задача при работе с ними:

Минимизировать количество эффектов и изолировать их от другого кода

Техника, которую мы можем использовать для управления эффектами, называется **функциональное ядро в императивной оболочке** или **Impureim Sandwich**.   При использовании этого подхода логику приложения мы описываем в виде чистых функций, а всё «нечистое» взаимодействие с внешним миром *отодвигаем к краям приложения*.

Receive Data
from Outer World
(Side Effect)

Transform Data
By the Business Rules
(Pure Functions)

Save Data
in Outer World
(Side Effect)

Получается такой бутерброд: эффект для получения данных; логика без
эффектов; эффект для сохранения данных

Рассмотрим подход на примере функции `updateUserInfo`. Сейчас преобразования данных в ней
перемешаны с их получением и сохранением:

JAVASCRIPT

```
function updateUserInfo(event) {  
    const { email, birthYear, password } = event.target;  
    const root = document.querySelector(".userInfo");  
  
    root.querySelector(".age").innerText = new Date().getFullYear() - birthYear;  
    root.querySelector(".password").innerText = password.replace(/\./g, "*");  
    root.querySelector(".login").innerText = email.slice(  
        0,  
        email.lastIndexOf("@")  
    );  
}
```

Попробуем отделить логику от эффектов. Сперва мы можем это сделать прямо внутри функции,
сгруппировав код в «кушки»:

```

function updateUserInfo(event) {
    // Получаем данные:
    const { email, birthYear, password } = event.target;

    // Преобразуем их:
    const age = new Date().getFullYear() - birthYear;
    const username = email.slice(0, email.lastIndexOf("@"));
    const hiddenPassword = password.replace(/./g, "*");

    // «Сохраняем», в нашем случае отображаем их в UI:
    const root = document.querySelector(".userInfo");
    root.querySelector(".age").innerText = age;
    root.querySelector(".password").innerText = hiddenPassword;
    root.querySelector(".login").innerText = username;
}

```

Затем мы можем выделить работу с данными в отдельную функцию. Она ничего не будет знать о получении и сохранении данных и будет заниматься только их преобразованиями:

```

function toPublicAccount({ email, birthYear, password, currentYear }) {
    return {
        age: currentYear - birthYear,
        username: email.slice(0, email.lastIndexOf("@")),
        hiddenPassword: password.replace(/./g, "*"),
    };
}

```

Тогда использовать функцию `toPublicAccount` внутри `updateUserInfo` можно будет таким образом:

```
function updateUserInfo(event) {  
    // Получаем данные:  
    const { email, birthYear, password } = event.target;  
    const currentYear = new Date().getFullYear();  
  
    // Преобразуем их:  
    const { age, username, hiddenPassword } = toPublicAccount({  
        email,  
        birthYear,  
        password,  
        currentYear,  
    });  
  
    // «Сохраняем»:  
    const root = document.querySelector(".userInfo");  
    root.querySelector(".age").innerText = age;  
    root.querySelector(".password").innerText = hiddenPassword;  
    root.querySelector(".login").innerText = username;  
}
```

Мы можем проверить, стал ли код лучше, если напишем тесты на преобразования данных. В первой версии кода, тест получился бы таким:

```
// 1. В случае с функцией `updateUserInfo`  
//    приходится создавать моки для DOM и Event:  
const dom = jsdom(<*>...*</>);  
const event = {  
  target: {  
    email: "test@test.com",  
    password: "strong-password-1234",  
    birthYear: 1994,  
  },  
};  
  
// ...Нужно создавать мок для текущей даты,  
// чтобы результат теста был воспроизводимым:  
jest.useFakeTimers().setSystemTime(new Date("2022-01-01"));  
  
// ...Также надо следить, чтобы все моки и таймеры  
//    сбрасывались и не влияли на другие тесты:  
afterAll(() => jest.useRealTimers());  
  
describe("when given a user info object", () => {  
  it("should calculate the user age", () => {  
    updateUserInfo(event);  
  
    // ...Проверять работу приходится по содержимому DOM-узла:  
    const node = dom.document.querySelector(".userInfo .age");  
  
    // ...Тип данных при этом теряется,  
    //    потому что DOM-узлы содержат только строки:  
    expect(node.innerText).toEqual("28");  
  });  
});
```

Теперь напишем тест для функции `toPublicAccount` и сравним с предыдущим:

```
// 2. В случае с `toPublicAccount` моки не нужны,
//      так как для тестирования чистой функции
//      нужны только входные данные и ожидаемый результат.

describe("when given a user info object", () => {
  it("should calculate the user age", () => {
    const { age } = toPublicAccount({
      email: "test@test.com",
      password: "strong-password-1234",
      birthYear: 1994,
      currentYear: 2022,
    });

    // Проверить результат мы можем прямым сравнением,
    // без моков, не потеряв при этом тип данных:
    expect(age).toEqual(28);
  });
});
```

В первом случае функция `updateUserInfo` занимается разными задачами: преобразованием данных и взаимодействием с UI. Её тесты это подтверждают: они проверяют, как изменились данные, но при этом создают моки для DOM.

При появлении другой похожей функции в её тестах пришлось бы мокать DOM *снова*, чтобы проверить изменение уже её данных. Это должно насторожить, потому что в тестах появляется дублирование, которое не несёт пользы.

Во втором случае тест проще, потому что ему не нужны моки и таймеры. Для тестирования чистых функций нам нужны лишь входные данные и ожидаемый результат. (Поэтому часто говорят, что чистые функции — тестируемы по своей природе.)

Взаимодействие с DOM при этом становится отдельной задачей. Моки для DOM будут появляться в тестах модуля, который будет заниматься взаимодействием с UI, и больше нигде.

Мы таким образом не только упростили код функции, но и улучшили разделение ответственности между разными частями приложения.

Уточню, что смысл не в том, будто «моки — это всегда плохо», нет. Иногда моки — единственный способ протестировать желаемый эффект, например, в адаптерах. ↗

Смысль в том, что если для тестирования логики приходится писать мок, то скорее всего есть способ упростить код, а эффекты вынести за пределы функции или модуля.

После рефакторинга мы можем заметить, что задача функции `updateUserInfo` превратилась в «композицию» функциональности. Она теперь собирает вместе получение, преобразование и сохранение данных.

Функция по структуре начала напоминать бутерброд из эффекта, логики и ещё одного эффекта — это и есть функциональное ядро в императивной оболочке. При адекватном разделении ответственности «слои» бутерброда будут независимы друг от друга. Это сделает работу с данными предсказуемой, а эффекты — изолированными и ограниченными.

Адаптеры для эффектов

Разделение логики и эффектов помогает обнаружить дублирование в получении и сохранении данных. Это может быть заметно по одинаковым мокам в тестах или похожему коду в самих эффектах.

Если разные части приложения общаются с внешним миром одинаково, мы можем поручить общение отдельной сущности — *адаптеру*. Адаптер уменьшит дублирование, отцепит код приложения от внешнего мира и сделает тестирование эффектов проще:

```
// Если мы замечаем одинаковую функциональность
// при получении или сохранении данных:

function updateUserInfo(user) {
    // ...

    if (window?.localStorage) {
        window.localStorage.setItem("user", JSON.stringify(user));
    }
}

function updateOrder(order) {
    // ...

    if (window?.localStorage) {
        window.localStorage.setItem("order", JSON.stringify(order));
    }
}

// Мы можем вынести её в адаптер:

const storageAdapter = {
    update(key, value) {
        if (window?.localStorage) {
            window.localStorage.setItem(key, JSON.stringify(value));
        }
    },
};

// И использовать уже только его:

function updateUserInfo(user) {
    // ...
    storageAdapter.update("user", user);
}

function updateOrder(order) {
    // ...
    storageAdapter.update("order", order);
}

// Теперь весь доступ к хранилищу описан в `storageAdapter`,
```

```
// и нам достаточно протестировать работу с хранилищем только в нём,  
// не дублируя проверки в функциях типа `updateUserInfo`.
```

ПОДРОБНЕЕ

Подробнее об адаптерах мы поговорим ещё отдельно в главе об архитектуре.

Команды и запросы

Когда мы отодвинули общение с внешним миром к краям приложения, мы можем подумать о его влиянии на внешний мир. Разные эффекты влияют на мир по-разному:

- одни — получают информацию из него;
- другие могут добавлять, обновлять и удалять её.

Разделение кода, ответственного за эти задачи, — это *разделение на команды и запросы (Command-Query Separation, CQS)*.[♂]

Согласно CQS запросы возвращают данные и не производят эффектов, а команды меняют состояние внешнего мира и ничего не возвращают. По сути цель CQS:

Разделить чтение и запись информации

Смешение команд и запросов запутывает код и делает его небезопасным. Сложно предсказать результат выполнения функции, если она может как-то поменять данные перед их получением или сразу после. При рефакторинге нам стоит обращать внимание на эффекты, которые нарушают CQS.

Для примера, посмотрим на сигнатуру функции `getLogEntry`:

TYPESCRIPT

```
function getLogEntry(id: Id<LogEntry>): LogEntry {}
```

Из типов мы можем сделать вывод, что эта функция каким-то образом *получает* данные из логов. Для нас станет сюрпризом, если в реализации мы увидим:

```
function getLogEntry(id: Id<LogEntry>): LogEntry {
    const entry =
        logger.getById(id) ?? logger.createEntry(id, Date.now(), "Access");

    return entry;
}
```

Проблема функции в её непредсказуемости. Мы не можем понять заранее, какой результат получим после вызова. Можно попробовать решить эту проблему, добавив деталей в имя функции:

```
function getOrCreateLogEntry(id: Id<LogEntry>): LogEntry {}
```

Информации стало больше, но мы всё ещё не можем узнать до вызова, будет создана новая запись в логах или будет найдена существующая.

Чем менее предсказуема функция, тем больше будет проблем с её отладкой. Отладка тем быстрее, чем меньше предположений нам необходимо проверить. Когда мы не можем предсказать поведение функции, нам требуется проверить *больше предположений* — это занимает больше времени.

Кроме этого непредсказуемые эффекты гораздо сложнее тестировать. ↗ Например, для проверки функции `getOrCreateLogEntry` нам бы пришлось написать какой-то такой тест:

```

afterEach(() => jest.clearAllMocks());
afterAll(() => jest.restoreAllMocks());

describe("when given an ID that exists in the service", () => {
  it("should return the entry with that ID", () => {
    jest.spyOn(logger, "getById").mockImplementation(() => testEntry);
    const result = getOrCreateLogEntry("test-entry-id");
    expect(result).toEqual(testEntry);
  });
});

describe("when given an ID of an entry that doesn't exist", () => {
  it("should create a new entry with the given ID", () => {
    jest.spyOn(logger, "getById").mockImplementation(() => null);
    const spy = jest.spyOn(logger, "createEntry");

    const result = getOrCreateLogEntry("test-entry-id");
    expect(spy).toHaveBeenCalledWith("test-entry-id", timeStub, "Access");

    expect(result).toEqual({
      createdAt: timeStub,
      id: "test-entry-id",
      type: "Access",
    });
  });
});

```

По количеству моков мы можем сделать вывод, что функция делает «слишком много». Сами моки при неаккуратном написании могут влиять на функциональность друг друга — это сделает тесты ненадёжными и хрупкими. Ну и идеально создание и чтение всё-таки разные операции.

Мы можем разделить эту функцию на две:

```

function readLogEntry(id: Id<LogEntry>): MaybeNull<LogEntry> {}
function createLogEntry(id: Id<LogEntry>): void {}

```

По сигнатурам мы теперь видим, что первая функция возвращает результат, а вторая — *что-то делает, но ничего не возвращает*. Уже это подталкивает к выводу, что вторая функция *меняет* состояние — то есть является эффектом.

```

function readLogEntry(id: Id<LogEntry>): MaybeNull<LogEntry> {
    return logger.getById(id) ?? null;
}

function createLogEntry(id: Id<LogEntry>): void {
    logger.createEntry(id, Date.now(), "Access");
}

```

Предсказуемость кода стала выше, потому что сигнатура функций перестала нас обманывать. Теперь она наоборот помогает предугадывать поведение ещё до того, как мы посмотрим на реализацию.

Тестирование обеих функций теперь будет независимым. Нам не потребуется мокать внутреннюю функциональность сервиса `logger`, чтобы проверить детали каждого эффекта. Достаточно будет проверить, что функции дёргают нужные эффекты с правильными данными:

```

// readLogEntry.test.ts

describe("when given an ID", () => {
    it("should call the logger service with that ID", () => {
        const spy = jest.spyOn(logger, " getById");
        readLogEntry("test-entry-id");
        expect(spy).toHaveBeenCalledWith("test-entry-id");
    });
});

// createLogEntry.test.ts

describe("when given an ID", () => {
    it("should call the logger service create with that ID and default entry data", () => {
        const spy = jest.spyOn(logger, "createEntry");
        createLogEntry("test-entry-id");
        expect(spy).toHaveBeenCalledWith("test-entry-id", timeStub, "Access");
    });
});

```

Часто для такого рода адаптеров нужно ещё протестировать, что они корректно преобразуют результат работы сервиса к нашим требованиям («адаптируют интерфейс»). Но конкретно в этом примере я посчитал это лишним и не стал заострять на этом внимания.

CQS и сгенерированные ID

В бекенду-разработке есть распространённый паттерн, который противоречит CQS. Его суть в том, что модуль работы с базой данных, в ответ на создание сущности возвращает сгенерированный для неё ID.

В целом, мне такое нарушение не кажется смертельным. Всё-таки CQS — это рекомендация, применимость которой стоит оценивать в каждой конкретной ситуации. Если возвращать ID — это общепринятый в проекте паттерн, в его использовании нет ничего страшного. Главное, чтобы оно было последовательным и задокументированным.

Если же от CQS отступать не хочется, можно передавать ID вместе с данными сущности, которые надо сохранить.

ПОДРОБНЕЕ

Подробно об этом решении писал Марк Симанн в статье “CQS versus server generated IDs”.[♂] В ней он детально объясняет само решение, его вариации, применимость и недостатки.

CQRS

Говоря о бекенде, стоит вспомнить о CRUD-операциях и CQRS.[♂] При проектировании API может возникнуть желание использовать одинаковые модели для чтения и записи данных:

TYPESCRIPT

```
type UserModel = {
    id: Id<User>;
    name: FullName;
    birthDate: DateTime;
    role: UserRole;
};

function readUser(id: Id<User>): UserModel {}
function updateUser(user: UserModel): void {}
```

В большей части случаев такое решение достаточно и проблем не вызовет. Однако, оно может стать проблемой, если мы читаем и пишем данные по-разному. Например, если мы хотим обновлять данные пользователя по частям.

Функция `updateUser` требует на вход весь объект `UserModel`, поэтому обновить отдельные поля мы не можем. Нам придётся передавать в функцию обновлённый объект целиком.

Если мы в проекте столкнулись с такой проблемой, то стоит вспомнить о *Command-Query Responsibility Segregation, CQRS*.⁹ Этот принцип расширяет идею CQS, предлагая использовать разные модели для чтения и записи данных. Продолжая пример с `UserModel`, мы можем выразить суть CQRS таким образом:

TYPESCRIPT

```
// Для чтения используем один тип, `ReadUserModel`:  
function readUser(id: Id<User>): ReadUserModel {}  
  
// Для записи – другой, `UpdateUserModel`:  
function updateUser(user: UpdateUserModel): void {}
```

Независимые модели «развязывают руки» в том, какие данные передавать при записи и какие данные ожидать при чтении. Например, мы можем описать тип `ReadUserModel` как набор обязательных полей, которые обязаны быть в получаемых данных:

TYPESCRIPT

```
type ReadUserModel = {  
    id: Id<User>;  
    name: FullName;  
    birthDate: DateTime;  
    role: UserRole;  
};
```

...И это не ограничит нас, если обновлять мы захотим только часть данных пользователя:

```

type UpdateUserModel = {
    // ID обязательен, чтобы было понятно,
    // данные какого пользователя обновлять:
    id: Id<User>;

    // Всё остальное опционально,
    // чтобы обновить только то, что нужно:
    name?: FullName;
    birthDate?: DateTime;

    // Роль, например, обновить вовсе нельзя,
    // поэтому этого поля здесь нет совсем.
};


```

БУДЬТЕ ВНИМАТЕЛЬНЫ

CQRS увеличивает количество кода (моделей, объектов, типов) в проекте. Перед использованием его стоит обсудить с командой и убедиться, что нет аргументов против.

Основные причины для использования CQRS — это отличия в структурах данных для чтения и записи, а также разница в нагрузке и необходимость в раздельном масштабировании чтения и записи.

В остальных случаях, вероятно, будет проще и дешевле использовать общую модель.

1. git-bisect, Use binary search to find the commit that introduced a bug, <https://git-scm.com/docs/git-bisect>
2. Referential Transparency, Haskell Wiki, https://wiki.haskell.org/Referential_transparency
3. "A pipe operator for JavaScript" by Axel Rauschmayer, <https://2ality.com/2022/01/pipe-operator.html>
4. "Functional Core in Imperative Shell" by Gary Bernhardt, <https://www.destroyallsoftware.com screencasts/catalog/functional-core-imperative-shell>
5. "Impureim Sandwich" by Mark Seemann, <https://blog.ploeh.dk/2020/03/02/impureim-sandwich/>
6. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
7. "Command-Query Separation" by Martin Fowler, <https://martinfowler.com/bliki/CommandQuerySeparation.html>

8. "CQS versus server generated IDs" by Mark Seemann, <https://blog.ploeh.dk/2014/08/11/cqs-versus-server-generated-ids/>
9. CRUD, Википедия, <https://ru.wikipedia.org/wiki/CRUD>
10. "Command-Query Responsibility Segregation" by Martin Fowler, <https://martinfowler.com/bliki/CQRS.html>

Обработка ошибок

Ошибки можно обрабатывать по-разному. Техника и стиль обработки будут зависеть от языка программирования, стиля кода, проекта и его ограничений, а часто — и личных предпочтений команды или конкретных людей.

В этой главе мы не будем стараться охватить все возможные варианты обработки ошибок. Вместо этого мы посмотрим на наиболее общие принципы и техники рефакторинга, которые могут помочь улучшить код.

УТОЧНЕНИЕ

У меня нет цели «продать» конкретный способ обрабатывать ошибки. Вместо этого я хочу показать, на что чаще всего обращаю внимание во время рефакторинга и какие обстоятельства могут усложнить обработку ошибок.

Я могу быть не прав. Описанные приёмы помогают лично мне, но это не значит, что они универсально применимы. Перед их использованием изучите, как обработка ошибок устроена в вашем проекте, обсудите идеи с командой и удостоверьтесь, что они подходят всем.

У любых инструментов есть область применения и ограничения, поэтому не используйте их только ради использования. Если ваш проект написан в ООП-стиле, будет странно тащить туда функциональное связывание, как бы заманчиво это ни было.

Грамотная обработка ошибок *помогает* справляться с нештатными ситуациями в приложении и дебажить проблемный код. Добиться этого можно разными способами, но как правило, удобная обработка ошибок начинается с 3 эвристик:

- Если ошибка мешает нормальному выполнению бизнес-логики — не пытаться продолжать работу, а переходить к её обработке.
- Обрабатывать ошибки централизованно, но не смешивать в кучу разные виды ошибок.
- Никогда не умалчивать никакие ошибки, особенно — необработанные.

В этой главе мы разберёмся, чем полезен каждый пункт и как их реализовать в коде. Но прежде, чем приступить к рефакторингу, немного договоримся о терминах и видах ошибок.

Виды ошибок

В "Domain Modeling Made Functional" Скотт Влашин делит ошибки на 3 вида:[♂]

- *Доменные*. Такие ошибки ожидаемы в бизнес-процессах, мы знаем их причину и как их обрабатывать. Например, деление на 0 в калькуляторе — это доменная ошибка, потому что запрет операции — это часть предметной области.
- *Инфраструктурные*. Тоже ожидаемы и понятны в обработке, но связаны с инфраструктурой, а не бизнес-логикой. Например, неудачный сетевой запрос.
- *Паники*. Неожиданные ошибки, мы не знаем как их обработать и восстановиться после них. Например, получить `null` там, где его быть не должно — это паника, потому что непонятно, как продолжать работу программы без нужных данных.

К слову

Я специально не использую термин «исключение», потому что в разных источниках «исключения» и «ошибки» значит ровно противоположное.^{♂ ♂} Вместо него я буду использовать термин «паника».

Мы будем часто сверяться с этим списком во время рефакторинга. Он поможет нам не смешивать обработку разных ошибок в кучу, а также выбирать подходящие техники обработки под конкретные случаи.

Техники обработки

Техники обработки ошибок зависят от языка, предпочтений команды, специфики и ограничений проекта. Чаще всего встречается код, где обработка ошибок построена одним из 4 способов:

- С помощью выбрасывания через `throw`;
- Использования Result-контейнеров;
- Сочетания выбрасывания и контейнеров;
- Сочетания контейнеров и функционального связывания.

Мы не будем «ранжировать» их «от плохих к хорошим». Вместо этого обсудим их преимущества и недостатки, а также исследуем, какие обстоятельства в проекте могут подтолкнуть к использованию той или иной техники.

Например, сперва мы обсудим, как рефакторить код, в котором можно использовать только выбрасывание паник. Затем поговорим, чем полезны контейнеры, как и когда можно использовать

их. Ближе к концу главы посмотрим, как можно сочетать разные техники вместе.

БУДЬТЕ ВНИМАТЕЛЬНЫ

Повторюсь, что приёмы в книге — это рекомендации, а не правила. Решение, применять ли эти идеи, стоит принимать в каждой конкретной ситуации, посоветовавшись с другими разработчиками.

Выбрасывание паник

В JavaScript-коде самый частый способ работы с ошибками — выбросить панику с помощью

`throw new Error()`. Выбрасывание работает нативно «из коробки», и этот способ — первый, который всплывает в голове, когда надо «сообщить, что что-то пошло не так».

К СЛОВУ

Кроме `Error` ещё есть `Promise.reject`, но он скорее связан с асинхронными операциями, поэтому, например, в бизнес-логике им пользуются гораздо реже.

Главный недостаток паник в том, что они больше подходят для неожиданных ошибок, чем для ожидаемых. Паники описывают ситуации, которые приводят программу в неконсистентное состояние, от которого *нельзя восстановиться*. От ожидаемых же ошибок восстановиться можно и известно как. Из-за этой разницы есть даже рекомендации избегать паник в бизнес-логике. ☹ ☹

На практике, однако, добиться такого разделения можно не всегда. Бывают проекты, в которых обработка паник и ошибок перемешана и устроена одинаково с помощью выбрасывания. В таких проектах может быть сложно внедрить альтернативные способы обработки ошибок и приходится работать только с паниками. Но даже такой код мы всё ещё можем сделать немного лучше.

Разберём на примере. Допустим, у нас есть функция `getUser`, которая обращается к API, чтобы получить данные о пользователе. После получения ответа она парсит его и сохраняет результат в хранилище.

JAVASCRIPT

```
async function getUser(id) {
  const dto = await fetchUser(id);
  const user = dto ? parseUser(dto) : null;
  if (user) storage.setUser(user);
  else storage.setError("Something went wrong.");
}
```

Функция `fetchUser` занимается запросом к сети и возвращает DTO из ответа сервера или `null`, если сервер ответил с ошибкой:

JAVASCRIPT

```
async function fetchUser(url) {  
    const response = await fetch(url);  
    const { value, error } = await response.json();  
    if (error) return null;  
    return value;  
}
```

Функция `parseUser` парсит серверный ответ и возвращает объект пользователя или `null`, если DTO оказался невалидным:

JAVASCRIPT

```
function parseUser(dto: UserDto): User | null {  
    if (!dto || !dto.firstName || !dto.lastName || !dto.email) return null;  
    return { ...dto, fullName: `${dto.firstName} ${dto.lastName}` };  
}
```

Чтобы понять, как именно рефакторить этот код, сперва определим в нём проблемы:

- Обработка ошибок как таковой тут нет. Мы возвращаем из функций `null`, когда что-то идёт не так, но умалчиваем причины ошибок и никак их не обрабатываем.
- Из-за `null` в результатах `fetchUser` и `parseUser` мы теряем контекст того, что именно пошло не так, и на уровне выше приходится проверять данные на те же ошибки снова. Из-за дублирования проверок код становится шумным.
- В функции `fetchUser` учтены только некоторые проблемы, а явного делегирования непредвиденных ошибок другим модулям нет. При исполнении такого кода приложение может упасть на любой строчке.
- Мы не различаем ошибки инфраструктуры и предметной области. При анализе ошибок эта информация могла бы понадобиться для поиска мест, где приложение сломалось.

Попробуем эти проблемы исправить.

Неожиданные ошибки и потеря контекста

Одна из причин для рефакторинга кода с выбрасыванием — неочевидность того, какие ошибки мы отловим, а какие нет. В таком коде легко пропустить необработанную ошибку, и приложение упадёт, когда мы этого не ожидаем.

Например, сейчас если внутри `fetchUser` произойдёт ошибка, которую мы не ждали, приложение упадёт:

```

async function fetchUser(url) {
    const response = await fetch(url);

    // Допустим, после распаковки JSON мы получили не объект, а `null`,
    // тогда следующая строка выбросит `null is not an object`:
    const { value, error } = await response.json();
    // ...
}

```

Мы можем это решить, добавив обработку через `try-catch` на уровне выше. Выброшенная ошибка будет перехвачена в функции `getUser`, и мы сможем её обработать:

```

async function getUser(id) {
    try {
        const dto = await fetchUser(id);
        const user = dto ? parseUser(dto) : null;
        if (user) storage.setUser(user);
        else storage.setError("Something went wrong.");
    } catch (error) {
        storage.setError("Couldn't fetch the data.");
    }
}

```

Однако, если мы выбрасываем паники для обработки всех ошибок, то мы можем случайно смешать ожидаемые и неожидаемые ошибки. Например, если валидация тоже выбрасывает паники:

```

function parseUser(dto: UserDto): User {
    if (!dto) throw new Error("Missing user DRO.");

    const { firstName, lastName, email } = dto;
    if (!firstName || !lastName || !email) throw new Error("Invalid user DRO.");

    return { ...dto, fullName: `${firstName} ${lastName}` };
}

```

То `try-catch` на уровне выше сможет их поймать, но мы не сможем отличить разные ошибки друг от друга:

```
async function getUser(id) {  
    try {  
        // ...  
    } catch (error) {  
        // error — это ошибка сети или ошибка валидации?  
        // Она ожидаема или нет?  
        //  
        // Мы можем определять ошибки по сообщению,  
        // которое передавали в конструктор Error,  
        // но это ненадёжно.  
    }  
}
```

ЗАЧЕМ ЗНАТЬ РАЗНИЦУ

Мы хотим различать ошибки и паники, потому что их может быть нужно по-разному обрабатывать. В зависимости от требований проекта может быть нужно, например, логировать паники в алёрт-мониторинге, а ожидаемые ошибки отражать в сервисе аналитики. Чем проще их отличить, тем меньше кода уйдёт на раздельную обработку.

Чтобы отличать паники от доменных и инфраструктурных ошибок, мы можем использовать для всех них отдельные типы.

Разные типы ошибок

В случае с JavaScript «отдельный тип» ошибки — это класс, который расширяет `Error`. В таких расширениях мы можем указать название или вид ошибки, а также какую-то дополнительную информацию в отдельных полях.

Например, для различения сетевых ошибок и ошибок валидации, мы можем создать такие типы:

```
// Ошибки валидации:
class InvalidUserDto extends Error {
    constructor(message) {
        super(message ?? "The given User DTO is invalid.");
        this.name = this.constructor.name;
    }
}

// Ошибки API:
class NetworkError extends Error {
    constructor(message, status, traceId) {
        this.name = this.constructor.name;
        this.message = message ?? messageFromStatus(status);

        // Можно расширить дополнительными полями для логирования:
        this.status = status;
        this.traceId = traceId;
    }
}
```

К СЛОВУ

Если при валидации надо выбросить несколько ошибок сразу, а не выбрасывать по одной, то можно расширить класс дополнительными полями или использовать `AggregateError`.

Тогда при отлове мы сможем по типу понять, что именно случилось:

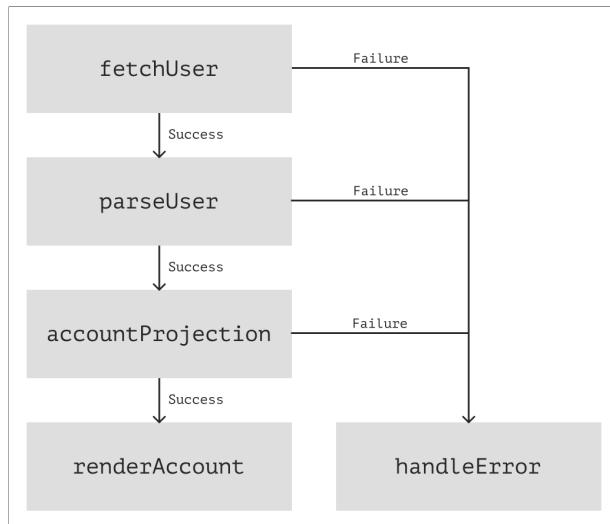
```
async function getUser(id) {
    try {
        ...
    } catch (error) {
        if (error instanceof InvalidUserDto) {
        } else if (error instanceof NetworkError) {
            ...
        } else throw error;
    }
}
```

Fail Fast

У такой обработки много минусов. Например, она использует выбрасывание в коде бизнес-логики, а также нарушает LSP⁸ внутри блока `catch`. Но несмотря на это у неё есть одно преимущество: при появлении ошибки мы не пытаемся продолжать работу, а переходим к её обработке.

Мы не хотим продолжать «нормальную работу», потому что ошибка — это *неконсистентное состояние* приложения. В таком состоянии нет гарантий, что данные приложения валидны, целостны, и с ними можно продолжать работать.

Если мы в этот момент завершаем выполнение и переходим к обработке, мы обезопасиваем себя от дальнейших ошибок, которые могут сильнее повредить данные. Кроме этого мы очищаем код доменной модели, освобождая его от лишних проверок.



Выполнение кода становится линейным, а все ветви с ошибками сразу же ведут в точку обработки

Rethrow

В последнем примере на последней строке блока `catch` появилось выражение `else throw error` — это так называемое *перебрасывание (rethrow)*. Мы используем его как механизм, который помогает *не замалчивать* ошибки, которые мы не можем обработать.

Если текущий обработчик проверил ошибку на все знакомые ему типы, но не нашёл подходящего, он может перебросить её на уровень вверх, чтобы ошибкой занялся «кто-то поумнее, кто знает, что делать».

Тут возникает вопрос, кто будет это обрабатывать. Об этом мы поговорим подробнее ближе к концу главы.

В таком коде функции «выпрямляются», а количество проверок на `null` уменьшается:

JAVASCRIPT

```
async function getUser(id) {
  try {
    const dto = await fetchUser(id);
    const user = parseUser(dto);
    storage.setUser(user);
  } catch (error) {
    if (error instanceof InvalidUserDto || error instanceof NetworkError) {
      storage.setError(error.message);
    } else throw error;
  }
}
```

Преимущества

Мы добились некоторых улучшений кода. И хоть их немного, они могут (не гарантируют, но могут) упростить работу:

- Выполнение стало более *линейным*. Мы перестали перепроверять данные на уже проверенные ошибки, поэтому поток кода стал более «прямым».
- Код переходит к обработке ошибок сразу, как только *нормальное выполнение становится невозможным*. Мы останавливаем ошибку в начале, не давая ей привести приложение в невалидное состояние.
- *Контекст ошибок сохраняется*, поэтому при начале обработки у нас есть полная информация о том, что произошло.

Проблемы

Но у выбрасывания есть и проблемы:

- Выброшенная ошибка может выстрелить в рантайме, если её не обработать.
- При этом в языке нет инструментов, которые бы заставили обрабатывать возможные ошибки, поэтому обработку легко забыть или пропустить.
- В коде почти нет синтаксической разницы между ошибками и паниками, из-за чего их сложно различать.
- При этом выбрасывать паники в коде доменной модели — явный запах, потому что ошибки домена — не паники, они ожидаются.
- Выполнить проверку на все потенциальные ошибки можно, но это выглядит некрасиво из-за `instanceof`.

- Использования `instanceof` можно избежать, создавая подклассы на каждый «слой» приложения, но это делает модель ошибок сложнее.
- Несяно, кто будет обрабатывать выброшенную ошибку. Полагаться на «договорённости» опасно, потому что в языке нет инструментов для принуждения к договорённостям.
- Нет чётких правил, которые бы однозначно указывали, когда и как оборачивать «низкоуровневый» код (`fetch`, Browser API и т.д.).
- Может пострадать производительность, потому что каждый объект `Error` собирает стек и другую информацию.
- Сигнатура функций не сообщает, что они могут выбросить ошибку — мы можем узнать об ошибках только из исходников.

УТОЧНЕНИЕ

Вообще, в TypeScript для предупреждения о возможных ошибках через сигнатуру функции можно использовать тип `never`¹⁰. Он не указывает, *какие именно* ошибки стоит ожидать, но как минимум намекает на их *возможность*. Это делает сигнатуру чуть точнее, но для более детального описания ошибок через типы `never` не подойдёт.

Если в проекте можно использовать *только* выбрасывание, то это, пожалуй, максимум, что мы можем сделать. По-хорошему, выбрасывать лучше только паники и избегать их в бизнес-логике. Но если в проекте есть ограничения, из-за которых нельзя использовать ничего другого, такой вид обработки может быть неплохим вариантом.

Однако, кроме выбрасывания есть и другие способы работы с ошибками. Если исследовав проект, мы заметили в коде какой-то намёк на *Result-контейнеры*, то функцию `getUser` можно улучшить.

Result-контейнеры

Как мы говорили ранее, некоторые ошибки ожидаемы. Например, мы можем ожидать, что API вернёт 404 в ответ на запрос несуществующей страницы. Такая ситуация нестандартная, но мы знаем, как её обрабатывать.

Для обработки ожидаемых ошибок может быть удобно использовать Result-контейнеры. Контейнер — это тип, который находится в одном из двух состояний: он либо содержит данные, если операция прошла без проблем, либо содержит возникшую ошибку. Схематично его можно изобразить так:

```
// Тип контейнера состоит из двух частей:  
// - Success – для ситуации, когда надо вернуть результат;  
// - Failure – для ситуации, когда надо вернуть ошибку.  
  
type Result<T0k, TErr> = Success<T0k> | Failure<TErr>;  
type Success<T> = { ok: true; value: T };  
type Failure<E> = { ok: false; error: E };
```

УТОЧНЕНИЕ

Реализация `Result` из примера намеренно абстрактная и не полная. Я сделал это, чтобы не претендовать на «каноничность».

Реализовать свой `Result`-контейнер с нуля — это интересная задача, но в продакшне я бы рекомендовал использовать известные сообществу решения. Например, для подобных задач подойдёт `Either` из библиотеки fp/ts. 

Если в вашем проекте уже используется некая реализация контейнеров, изучите её возможности. Вероятно, у неё уже есть всё необходимое.

Используя контейнер, мы могли бы переписать функцию `parseUser` как-то так:

```
type MissingDtoError = "MissingDTO";  
type InvalidDtoError = "InvalidDTO";  
type ValidationError = MissingDtoError | InvalidDtoError;  
  
function parseUser(dto: UserDto): Result<User, ValidationError> {  
    if (!dto) return Result.failure("MissingDTO");  
  
    const { firstName, lastName, email } = dto;  
    if (!firstName || !lastName || !email) {  
        return Result.failure("InvalidDtoError");  
    }  
  
    return Result.success({ ...dto, fullName: `${firstName} ${lastName}` });  
}
```

Теперь функция возвращает коробочку с результатом или ошибкой. Эта коробочка инкапсулирует информацию о произошедшей ситуации и возвращает её на уровень выше.

Сигнатура точнее отражает процесс

С контейнером мы видим возможные ошибки прямо в сигнатуре функции. Нам не требуется изучать исходники, чтобы узнать, какие ошибки мы ожидаем. Все ожидаемые исходы операции отражены в возвращаемом типе.

Использование контейнеров имеет и обратную сторону, конечно. Чтобы использовать данные из результата на уровне выше, нам надо контейнер «распаковать»:

TYPESCRIPT

```
async function getUser(id) {
    try {
        const { value: dto, error: networkError } = await fetchUser(id);
        const { value: user, error: parseError } = parseUser(dto);
        storage.setUser(user);
    } catch (error) {
        // ...
    }
}
```

Если произошла ошибка, то при распаковке значение `value` будет пустым, а значит передать данные в следующую функцию не получится. Так мы видим, что дальнейшая работа возможна, только если ошибок не было или они обработаны. То есть сигнатура контейнеров заставляет помнить о потенциальных ошибках и обрабатывать их:

TYPESCRIPT

```
async function getUser(id) {
    try {
        const { value: dto, error: networkError } = await fetchUser(id);
        // Обработать `networkError`...

        const { value: user, error: parseError } = parseUser(dto);
        // Обработать `parseError`...

        storage.setUser(user);
    } catch (error) {
        // Обработать непредвиденные ситуации...
    }
}
```

Для описания ожидаемых ошибок контейнеры, вероятно, подойдут лучше, чем паники. Они отличаются синтаксически, что помогает воспринимать их как именно ожидаемые. Кроме этого контейнеры не надо «выбрасывать», а значит они не сломают приложение.

В Node.js во времена колбеков опасные операции использовали нечто похожее на контейнер — кортеж из ошибки и значения: ⚡

```
function errorFirstCallback(err, value) {}
```

Переменная `err` пуста, если операция прошла успешно, или содержит возникшую ошибку. Это также известно как error-first callback.

Явная обработка

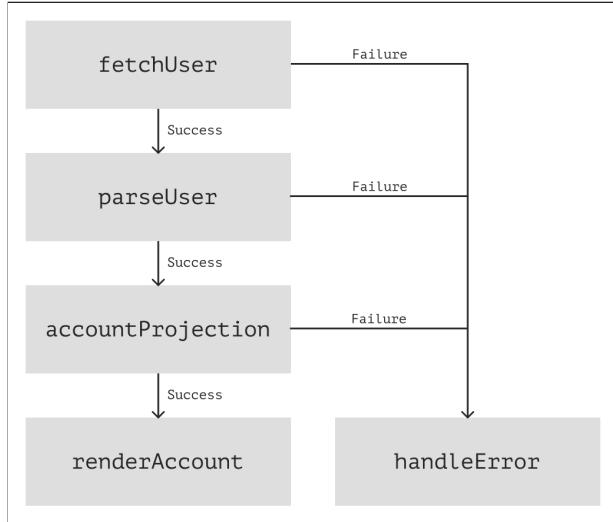
После распаковки контейнеров мы можем настроить обработку так, чтобы ошибки заканчивали «нормальное» выполнение программы — получим Fail Fast: ⚡

TYPESCRIPT

```
async function getUser(id) {
  try {
    const { value: dto, error: networkError } = await fetchUser(id);
    if (networkError) return handleError(networkError);
    // Или throw new Error(networkError) и обработать позже.

    const { value: user, error: parseError } = parseUser(dto);
    if (parseError) return handleError(parseError);

    storage.setUser(user);
  } catch (error) {
    // ...
  }
}
```



Цепочка результатов прервётся там, где появится ошибка, и управление перейдёт в конец

УТОЧНЕНИЕ

В императивном коде при большом количестве операций это будет выглядеть громоздко. Но если количество операций небольшое (1–2), то в целом это не так страшно. О том, как распаковывать контейнеры более изящно, мы поговорим чуть позже.

Централизованная обработка

Несмотря на необходимость обрабатывать ошибки явно, мы всё ещё можем настроить централизованную обработку. Как вариант, мы можем использовать отдельную функцию, которая будет заниматься обработкой ошибок и использовать её при распаковке контейнеров:

```
type ValidationError = MissingDtoError | InvalidDtoError;
type NetworkError = BadRequest | NotFound | ServerError;
type UseCaseError = ValidationError | NetworkError;

// Функция `handle GetUserErrors` обрабатывает ошибки юзкейса `getUser`:
function handleGetUserErrors(error: UseCaseError): void {
    const messages: Record<UseCaseError, ErrorMessage> = {
        MissingDTO: "The user DTO is required.",
        InvalidDTO: "The given DTO is invalid.",
        // `Record<UseCaseError, ErrorMessage>` удостоверится,
        // что мы покрыли все ожидаемые ошибки.
    };

    // Если ошибка неизвестная, перебросим её наверх:
    const message = messages[error];
    if (!message) throw new Error("Unexpected error when getting user data.");

    // Если ожидаемая, обработаем:
    storage.setError(message);

    // Если необходимо, добавим инфраструктурной функциональности:
    logger.logError(error);
    analytics.captureUserActions();
}
```

Паники отдельно

Так как «низкоуровневый» код не возвращает контейнер, а выбрасывает паники, нам надо оборачивать такие операции в **try-catch**, чтобы в случае проблемы вернуть контейнер:

```
type NetworkError = BadRequest | NotFound | InternalError;

async function fetchUser(url) {
  try {
    const response = await fetch(url);
    const { value, error } = await response.json();
    return error ? Result.failure("BadRequest") : Result.success(value);
  } catch (error) {
    //
    // Если ошибка ожидаема — возвращаем контейнер:
    const reason = errorFromStatus(error);
    if (reason) return Result.failure(reason);
    //
    // Если не ожидаема, то используем rethrow:
    else throw new Error("Unexpected error when fetching user data.");
  }
}
```

Обернуть таким образом нужно будет *каждый* вызов «низкоуровневого» API — в нашем случае *каждый* запрос к сети. Это может увеличить количество кода. Однако, если схема работы с этими API одинаковая, мы можем уменьшить дублирование с помощью декораторов:

```
// Декоратор будет принимать «опасную» функцию,
// и вызывать её внутри `try-catch`.
// При возникновении ошибки, будет проверять,
// ожидаема ли ошибка и возвращать контейнер
// или перебрасывать ошибку выше.

function robustRequest(request) {
    return async function perform(...args) {
        try {
            return await request(...args);
        } catch (error) {
            const reason = errorFromStatus(error);
            if (reason) return Result.failure(reason);
            else throw new Error("Unexpected error when making a request.");
        }
    };
}

// Использовать декоратор можно будет
// с разными функциями работы с сетью:

const safeFetchUser = robustRequest(fetchUser);
const safeCreatePost = robustRequest(createPost);
```

Множественные ошибки

Дополнительные данные, например для множественных ошибок, мы можем хранить в поле `error` контейнера. Мы можем передавать в этом поле объекты, массивы или даже `Error`-инстансы. Это может пригодиться, например, для ошибок валидации, которые содержат список невалидных полей.

К СЛОВУ

Один из способов сообщить о множественных ошибках — это паттерн «Уведомление» (Notification). ↪ ↪ Его тоже можно воспринимать, как один из видов контейнеров.

Распаковка

Главная проблема, конечно, никуда не делась — нам всё ещё нужно распаковывать контейнеры и прерывать выполнение кода вручную. Сделать это пару раз не сложно, но более частая ручная распаковка может сделать код шумным. На помощь в такой ситуации может прийти функциональное связывание.

Связывание результатов

ПРЕЖДЕ, ЧЕМ НАЧАТЬ

О функциональном связывании хорошо написал Скотт Влашин в посте о “Railway Oriented Programming”,[♂] советую прочесть эту статью перед продолжением. Я не буду подробно вдаваться в детали. В этой статье всё описано гораздо лучше, чем смог бы написать я.

КРОМЕ ЭТОГО

Если в вашем проекте используется ручная распаковка, и вас всё устраивает, то не стоит переписывать её на функциональное связывание. Оно может усложнить код, не принеся пользы.

О связывании стоит задуматься, если ваш проект написан в функциональном стиле, а ручная распаковка контейнеров начинает напрягать. В остальных случаях оно, скорее всего, не нужно.

Основная идея связывания в том, чтобы заставить контейнер самостоятельно заботиться о последовательном вызове нескольких функций. Сейчас вызвать функции напрямую одну за одной нельзя — входы и выходы функций не совместимы друг с другом:

TYPESCRIPT

```
function fetchUser(id: UserId): Result<UserDTO, FetchUserError> {}
function parseUser(dto: UserDTO): Result<User, ParseUserError> {}

// Мы хотим вызывать функции в виде:
// fetchUser -> parseUser -> storage.setUser

// Но сейчас этого сделать нельзя, потому что выход одной функции
// не соответствует входному типу следующей за ней.
//
// Функция `fetchUser` возвращает `Result<UserDTO, FetchUserError>` ,
// но `parseUser` на вход требует просто `UserDTO`.
```

Мы можем воспринимать связывание, как «подгонку» входного типа функций, чтобы они могли принимать на вход `Result<T0k, TErr>`. Если бы мы «подгоняли» типы руками, то получилось бы что-то типа:

```
function parseUser(
  result: Result<UserDTO, FetchUserError>
): Result<User, ParseUserError> {}
```

Но это не вариант, потому что мы не хотим на входе `parseUser` зависеть от результата конкретной предыдущей операции. Мы хотим уметь получать любые результаты и распаковывать автоматически.

Для автоматизации этого «железнодорожное программирование» предлагает использовать специальные функции-адаптеры. В зависимости от реализации контейнера, эти адаптеры могут быть отдельными функциями или методами контейнера. Я не буду претендовать на каноничность, поэтому моя реализация будет схематичной:

```
const Result = {
  // ...

  // Если текущий контейнер в состоянии Success,
  // то мы «распаковываем» значение и передаём его на вход функции fn.
  // Если контейнер в состоянии Failure,
  // возвращаем новый контейнер Failure<E> с ошибкой,
  // которая была в этом контейнере.
  bind: (fn) => (ok ? fn(value) : failure(error)),

  // При завершении цепочки выполнения
  // нам надо обработать ошибку или результат,
  // поэтому две функции-аргумента в этом методе:
  match: (handleError, handleValue) =>
    ok ? handleValue(value) : handleError(error),
};

// Метод `bind` будет «связывать» функции в цепочку,
// а метод `match` будет «разворачивать» финальный результат.
//
// Метод `match` принимает 2 аргумента:
// - первая функция – это обработчик ошибок, она будет обрабатывать ошибки всей цепочки.
// - а вторая – обработчик результата цепочки.
```

Теперь результаты можно связывать в виде цепочки из нескольких вызовов:

```
async function getUser(id) {
  (await fetchUser(id))
    .bind(parseUser)
    .match(handleGetUserErrors, storage.setUser);
}
```

В такой функции ошибка на любом из результатов выполнения сразу же отправится в функцию

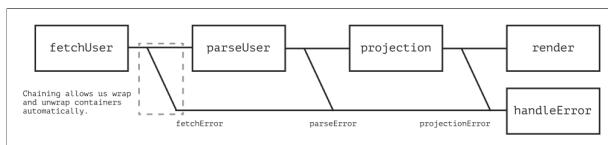
`handleGetUserErrors`. Так в сочетании с Exhaustiveness Check[⌚] мы можем организовать обработку всех ошибок для `getUser` без необходимости руками распаковывать контейнеры.

К СЛОВУ

Связывание может выглядеть по-разному. Это может быть цепочка методов, может быть функция типа `pipe`[⌚] которая «проталкивает» по себе результаты от одной функции к другой. Финальная обработка результатов тоже может выглядеть по-разному в зависимости от стиля и парадигмы проекта.

Если в вашем проекте есть контейнеры, приглядитесь к их реализации внимательнее, возможно, там уже есть всё нужное. Если нет, посмотрите на существующие решения типа `fp/ts` или `sweet-monads` и выберите себе ту, которая больше нравится.^{⌚ ⌚ ⌚}

Смысл связывания в *автоматизации «проталкивания»* и распаковки результатов. Мы соединяем все результаты в последовательность «рельсиков». Ошибка из любой функции съезжает по развилике на «колесо с ошибками» и катится по ней до самого конца. Это похоже на соединение железнодорожных путей — поэтому и программирование “railway oriented” 😊



Связывание помогает адаптировать входы и выходы функций

▶ Нативное связывание в JavaScript?..

Проблемы

Проблемы у связывания тоже есть:

- Код становится сложнее, это может увеличить порог входа в проект.
- Связывание требует договорённостей: нужно следить за использованием контейнеров и обёртками для «опасных» низкоуровневых функций.
- В нефункциональном коде это может выглядеть неоднозначно.
- В JavaScript не хватает нативного паттерн-матчинга для удобной работы с похожими идеями.

Когда предпочтеть паники

Связывание и контейнеры не могут полностью заменить паники, это немного разные инструменты. Поэтому паники могут быть полезны и при работе с контейнерами тоже.

Например, если контейнер никто не собирается обрабатывать или если не важно, какая произошла ошибка, то контейнер может быть не нужен. Хороший чеклист с тем, когда предпочесть контейнер, а когда панику вы сможете найти у Скотта Влашина в статье "Against Railway-Oriented Programming".[♂]

Cross-Cutting Concerns

Когда обработка ошибок описана единообразно во всём проекте, нам проще отыскать место с ошибкой по, например, баг-репорту. Но кроме этого централизованная обработка позволяет удобно компоновать cross-cutting concerns,[♂] например, логирование.

Если обработчики ошибок изолированы, их можно декорировать дополнительной функциональностью.[♂] Например, мы можем добавить логирование в обработчики ошибок с помощью декораторов:

```
// services/logger.js
// Сервис логирования предоставляет функцию
// для отправки нового события:

const logEvent = (entry: LogEntry) => {};

// infrastructure/logger.js
// Чтобы не добавлять логирование в каждый обработчик отдельно,
// мы можем создать декоратор, который будет принимать функцию,
// вызывать её и логировать результат вызова:

const withLogger =
  (fn) =>
  (...args) => {
    const result = fn(...args);
    const entry = createEntry({ args, result });
    logEvent(entry);
  };

// network.js
// Тогда для логирования ошибок будет достаточно
// «обернуть» обработчик в логирующий декоратор:

const handleNetworkError = (error: NetworkError) => {};
const errorHandler = withLogger(handleNetworkError);
```

В целом, cross-cutting concerns удобно оформлять в виде декораторов: так «сервисная» функциональность оказывается изолирована, не смешивается с остальным кодом приложения, а сами декораторы удобно «внедрять» в самые разные части кода.

УТОЧНЕНИЕ

У декораторов есть ограничения, и они подойдут не во всех случаях. Например, если нам нужно отправить сообщение в лог по середине работы функции, то, вероятно, декоратором этого добиться будет сложно.

Хотя в таких случаях обычно стоит подумать, почему нам нужно логировать что-то *по середине* функции, стоит ли разбить эту функцию на несколько. Но опять же, проекты и специфика бывают разными.

Зоопарк обработчиков

Потенциально опасные функции могут отличаться и быть связаны с разными API. Требования к обработке ошибок в таких функциях могут отличаться. Например, какие-то API вместо использования `try-catch` требуют указывать свойство `.онегигт` или слушать событие ошибки.

Централизованная обработка помогает решить эту проблему, потому что отцепляет *сами обработчики от места их использования*. Один обработчик становится возможно использовать с разными API и обрабатывать им ошибки от разных модулей.

К слову

Если сигнатура обработчика ошибки и API сервиса несовместимы, мы можем воспользоваться адаптером для решения этой проблемы. ☺

Иерархия отлова ошибок

Ранее мы упоминали перебрасывание ошибок (`rethrow`) — то есть делегирование их «вышестоящему» обработчику, если текущий не знает, как их обработать.

Делегирование создаёт иерархию обработчиков. Как с любыми иерархиями, её лучше держать как можно более плоской. Не всегда этого получится добиться, но всё же чем иерархия плосче, тем проще с ней работать. Один из вариантов организации обработчиков можно собрать из 3 уровней:

Обёртки над «низкоуровневым» кодом

На «низком» уровне мы оборачиваем в `try-catch` браузерные API, функции для работы с сетью, общения с девайсом и т.д.

Функции-обёртки ловят паники низкоуровневого кода. В зависимости от стиля обработки они далее конвертируют пойманные паники в контейнеры или перебрасывают их на уровень выше.

Обработчики пользовательских сценариев

На этом уровне мы обрабатываем ошибки пользовательских сценариев: ошибки доменной логики и ожидаемые инфраструктурные ошибки низкого уровня.

Такие обработчики могут ловить ошибки как одного сценария, так и набора связанных фич. Первые могут называться обработчиками ошибок юзкейса, а вторые — обработчиками слайса приложения.

Удобство обоих вариантов зависит от многих факторов и рекомендовать конкретный довольно сложно, поэтому в этой главе мы просто объединим их в «один уровень».

Сообщать пользователю о проблемах обычно проще всего именно отсюда, потому что на этом уровне есть доступ к сервисам, которые могут, например, вывести сообщение на экран или отправить запрос в алёрт-мониторинг. Не во всех проектах это будет самое удобное место, но чаще всего основу обработки удобно выстроить на этом уровне.

Обработчики последней надежды

На уровне всего приложения или веб-страницы мы отлавливаем все не обработанные ранее ошибки и паники.

Здесь полезно добавить логирование и инструменты диагностики, чтобы при анализе было проще понять, что привело к проблеме. Работать на этом уровне удобнее всего с паниками, потому что у них есть стек-трейс, по которому в будущем проще анализировать причины возникновения проблемы.

НАПРИМЕР

В React для обработки ошибок пользовательских сценариев и обработки последней надежды могут пригодиться Error Boundaries. ↗ Они отлавливают ошибки во время рендеря и выводят UI, где разработчики могут сообщить пользователю о проблеме.

КРОМЕ ЭТОГО

В JavaScript мы можем отловить необработанные ошибки на уровне глобального объекта с помощью специальных событий. ↗ ↗ ↗ Такие события обычно содержат информацию о причине и месте возникновения ошибки, поэтому их обработку полезно сочетать с логированием и инструментами алёрт-мониторинга.

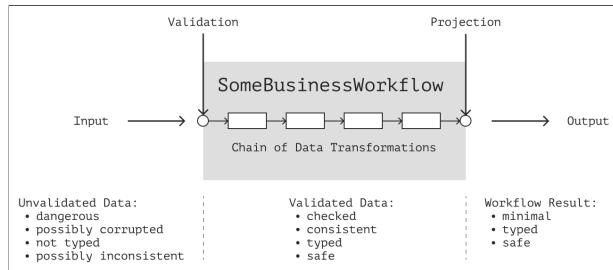
Предвалидация данных

В главе о функциональном пайплайне мы упоминали предвалидацию данных на входе в приложение перед их использованием. Её можно рассматривать, как часть обработки ошибок, потому что она прерывает выполнение программы при появлении невалидных данных.

Предвалидация освобождает код бизнес-логики от лишних проверок и позволяет сосредоточиться на преобразованиях данных в бизнес-процессах. Так из доменной модели пропадают лишние проверки, которые не имеют отношения к бизнес-процессам.

С предвалидацией все ошибки данных собираются в одном месте, а их обработка становится гибче. Например, с ней проще компоновать ошибки в наборы и обрабатывать их разом, а не по одной.

При рефакторинге обработки ошибок мы также можем пользоваться этой идеей и отодвигать проверки данных ближе ко входу в приложение:



Снаружи данные опасные, внутри — безопасные; наличие невалидных данных завершит выполнение и передаст управление обработчику ошибок

ПОДРОБНЕЕ

О предвалидации, поствалидации, селекторах и безопасности данных мы поговорим более детально в главе об архитектуре.

1. "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
2. "Errors Are Not Exceptions" by swyx, <https://www.swyx.io/errors-not-exceptions>
3. "The Error Model" by Joe Duffy, <http://joeduffyblog.com/2016/02/07/the-error-model/>
4. "Replacing Throwing Exceptions with Notification in Validations" by Martin Fowler, <https://martinfowler.com/articles/replaceThrowWithNotification.html>
5. "The Pragmatic Programmer" by Andy Hunt, https://www.goodreads.com/book/show/4099.The_Pragmatic_Programmer
6. `AggregateError`, MDN, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AggregateError
7. "A behavioral notion of subtyping" by Barbara H. Liskov, Jeannette M. Wing, <https://dl.acm.org/doi/10.1145/197320.197383>

8. More on Functions, `never`, TypeScript: Documentation, <https://www.typescriptlang.org/docs/handbook/2/functions.html#never>
9. fp-ts, Typed functional programming in TypeScript, <https://github.com/gcanti/fp-ts>
10. Error-first callbacks, Node.js Documentation, <https://nodejs.org/api/errors.html#error-first-callbacks>
11. Fail-fast, Wikipedia, <https://en.wikipedia.org/wiki/Fail-fast>
12. "Notification" by Martin Fowler, <https://martinfowler.com/eaaDev/Notification.html>
13. "Railway Oriented Programming" by Scott Wlaschin, <https://fsharpforfunandprofit.com/rop/>
14. `switch-exhaustiveness-check`, ES Lint TypeScript, <https://typescript-eslint.io/rules/switch-exhaustiveness-check/>
15. sweet-monads, Easy-to-use monads implementation with static types definition, <https://github.com/JSMonk/sweet-monads>
16. neverthrow, Type-Safe Errors for JS & TypeScript, <https://github.com/supermacro/neverthrow>
17. `pipe`, fp-ts, <https://gcanti.github.io/fp-ts/modules/function.ts.html#pipe>
18. "Against Railway-Oriented Programming" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/>
19. Cross-Cutting Concern, Wikipedia, https://en.wikipedia.org/wiki/Cross-cutting_concern
20. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
21. Decorator Pattern, Refactoring Guru <https://refactoring.guru/design-patterns/decorator>
22. Adapter Pattern, Refactoring Guru, <https://refactoring.guru/design-patterns/adapter>
23. Предохранители в React, <https://ru.reactjs.org/docs/error-boundaries.html>
24. Window: `error` event, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/API/Window/error_event
25. Window: `unhandledrejection` event, MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/API/Window/unhandledrejection_event
26. "Dealing with Unhandled Exceptions", by Alexander Zlatkov <https://blog.sessionstack.com/how-javascript-works-exceptions-best-practices-for-synchronous-and-asynchronous-environments-39f66b59f012#ecc9>

Интеграция модулей

Сложные приложения состоят из множества частей. Взаимодействие частей между собой влияет на организацию и сложность кода. Чем взаимодействие проще и очевиднее, тем легче читать и изменить код приложения. В этой главе мы рассмотрим, как замечать чрезмерно запутанную организацию кода и как упрощать компоновку приложения.

Зацепление и связность

Плохой код заметен по страху его менять. Страх возникает, когда нам кажется, что после изменений «всё развалится» или «придётся обновлять кучу другого кода». Такие ощущения возникают, если модули приложения слишком много знают друг о друге.

К СЛОВУ

Под модулем мы будем понимать обособленную часть приложения, которая отвечает за конкретную задачу и общается с внешним миром или другими модулями через публичное API.

Когда от одного модуля зависит внутреннее устройство других, изменения в нём влияют на остальные модули тоже. Остановить распространение изменений по кодовой базе в таком случае становится трудно. Работа над задачей, начинает затрагивать код, который к ней не относится. Из-за этого становится страшно вносить изменения, потому что сломаться может что угодно и необходимо перепроверять работу всего приложения.

Степень знания одного модуля об устройстве других называется *зацеплением (coupling)*.[♂] Чем выше зацепление, тем сложнее вносить изменения изолированно в конкретный модуль.

Разделение ответственности

В хорошо организованном приложении работа над задачей вызывает изменения только в коде, который связан с этой задачей. Этот принцип известен, как *разделение ответственности (Separation of Concerns, SoC)*.[♂]

Этот принцип помогает ограничивать распространение изменений по кодовой базе. Если весь код, ответственный за одну задачу, находится в одном модуле, то и изменения этой задачи будут влиять только на этот модуль. Всё, что к задаче не относится, находится за пределами модуля и не повлияет на его код.

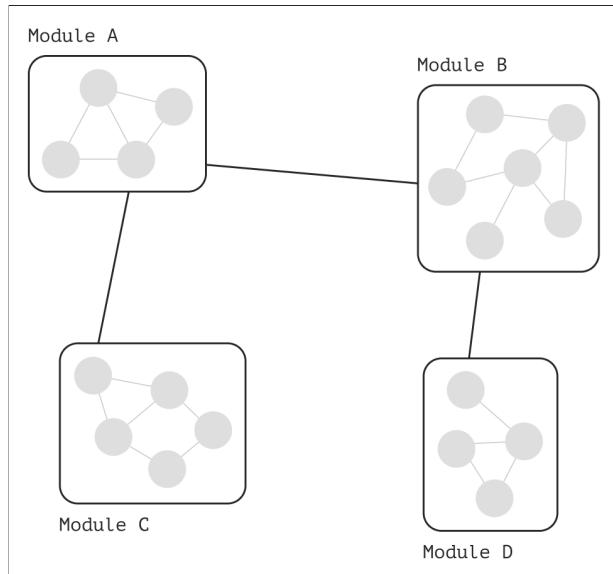
Степень «соответствия» кода задаче называется *связностью* (*cohesion*). [♂] Чем выше связность модуля, тем ближе его код по смыслу к задаче, ради которой его написали и тем проще отыскать его в кодовой базе.

Правило интеграции модулей

Первое и главное, что нам стоит проверить при анализе взаимодействия модулей во время рефакторинга, это правило:

Зацепление должно быть низким, а связность — высокой

Скомпонованная по такому правилу программа выглядит как «островки задач», связанные «мостиками» публичного API, событий или сообщений:



«Островки» отвечают за конкретные задачи предметной области и общаются друг с другом по «мостикам» публичного API, событий или сообщений

Разделение задач

Чтобы понять, по каким признакам во время рефакторинга искать слабое разделение задач, рассмотрим пример. Модуль `purchase` из фрагмента ниже сильно зацеплен с модулем `cart`. Он

использует внутренние детали объекта корзины (структуру объекта и тип поля `products`), чтобы узнать, пустая ли она:

TYPESCRIPT

```
// purchase.ts

async function makePurchase(user, cart) {
  if (!cart.products.length) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  await sendOrder(order);
}
```

Проблема этого кода в нарушении инкапсуляции. Модуль `purchase` не знает и не должен знать, как правильно проверить, пуста ли корзина.

Детали проверки корзины, не входят в задачу оформления заказа. Нам важен *факт*, что корзина непустая, но не важно, *как* этот факт будет определён. Реализация проверки — это задача модуля корзины, потому что именно он создаёт этот объект и знает, как держать его валидным:

TYPESCRIPT

```
// cart.ts
// Выносим проверку пустоты корзины в модуль `cart`:
export function isEmpty(cart) {
  return !cart.products.length;
}

// purchase.ts
import { createOrder } from "./order";
import { isEmpty } from "./cart";

async function makePurchase(user, cart) {
  if (isEmpty(cart)) throw new Error("Cart is empty!");

  const order = createOrder(user, cart);
  await sendOrder(order);
}
```

Теперь, если внутренняя структура корзины по каким-то причинам изменится, изменения ограничатся модулем `cart`:

```
// cart.ts
type Cart = {
    // Было products, стало items:
    items: ProductList;
};

export function isEmpty(cart) {
    // Место, которое надо поправить:
    return !cart.items.length;
}

// purchase.ts
async function makePurchase(user, cart) {
    if (isEmpty(cart)) throw new Error("Cart is empty!");
    // Использование в других модулях осталось неизменным,
    // мы ограничили распространение изменений.
}
```

...Модули, которые используют функцию `isEmpty` из публичного API, останутся неизменными. Если бы модули использовали устройство корзины напрямую, то при изменении свойства, пришлось бы обновлять их все.

Как определить связность

Часто с первого взгляда непонятно, относится задача к конкретному модулю, или нет. Для проверки гипотез об этом мы можем обращать внимание на данные, с которыми работает модуль или функция.

Данные — это *входные и выходные параметры*, а также *зависимости и контекст*, которые использует модуль. Чем меньше данные одного модуля схожи с данными другого модуля, тем выше вероятность, что они относятся к разным задачам. Если, например, функция часто работает с данными из соседнего модуля, скорее всего, она должна быть его частью.

К СЛОВУ

Мы можем знать эту проблему как запах кода “Feature Envy”. ☺

Представим, что мы рефакторим приложение для трекинга расходов. Допустим, в модуле, отвечающем за бюджет, мы видим такой код:

```
// budget.js

// Создаёт новый бюджет:
function createBudget(amount, days) {
  const daily = Math.floor(amount / days);
  return { amount, days, daily };
}

// Считает, сколько потрачено всего:
function totalSpent(history) {
  return history.reduce((tally, record) => tally + record.amount, 0);
}

// Добавляет трату, уменьшая доступное количество денег
// и создавая новую запись в истории трат:
function addSpending(record, { budget, history }) {
  const newBudget = { ...budget, amount: budget.amount - record.amount };
  const newHistory = [...history, record];

  return {
    budget: newBudget,
    history: newHistory,
  };
}
```

Задача модуля `budget` — отвечать за преобразования бюджета. Однако мы видим функции, которые работают не только с ним:

- Функция `totalSpent` работает только с историей записей о расходах;
- Функция `addSpending` работает с бюджетом, но тоже использует данные истории расходов.

Из данных, с которыми работают эти функции, мы можем сделать вывод, что они не так уж и относятся к бюджету. Например, `totalSpent` — больше относится к истории расходов, а `addSpending` — больше похожа на целый пользовательский сценарий приложения.

Попробуем распилить код по фичам, выделив историю и юзкейс в отдельные модули:

```
// budget.js
// Тут держим только код,
// отвечающий за преобразования бюджета:

function createBudget(amount, days) {
  const daily = Math.floor(amount / days);
  return { amount, days, daily };
}

function decreaseBy(budget, record) {
  const updated = budget.amount - record.amount;
  return { ...budget, amount: updated };
}

// history.js
// Здесь только код, отвечающий
// за преобразования истории трат:

function totalSpent(history) {
  return history.reduce((tally, record) => tally + record.amount, 0);
}

function appendRecord(history, record) {
  return [...history, record];
}

// addSpending.js
// Здесь описываем юзкейс добавления траты:
// - уменьшаем бюджет;
// - добавляем запись в историю.

function addSpending(spending, appState) {
  const budget = decreaseBy(state.budget, spending);
  const history = appendRecord(state.history, spending);
  return { budget, history };
}
```

В начале жизни приложения подобное строгое разделение модулей и фич может быть не нужно. Но если приложение надо масштабировать и добавлять больше пользовательских сценариев, вероятно, потребуется совмещать функциональность разных модулей. Нечёткое разделение модулей может привести к неявным зависимостями между ними, из-за которых компоновать функциональность будет сложнее.

Для беспроблемного масштабирования нам стоит следить за зацеплением и связностью. Если мы знаем, что будем расширять и переиспользовать функциональность, то лучше разделить код по разным модулям так, чтобы между ними было минимум скрытых зависимостей.

Контракты

Публичное API модуля можно назвать простой формой *контракта*.[⊗] Контракты фиксируют гарантии одной сущности перед другими: они требуют определённых аргументов и обязуют функции возвращать конкретный результат. Это позволяет другим частям программы полагаться не на устройство модуля, а только на его «обещания» и строить свою работу исходя из них.

Рассмотрим на примере, зачем это нужно. В коде ниже мы опираемся на устройство модуля `api`, тем самым повышая зацепление:

TYPESCRIPT

```
// ...
await api.post(api.baseUrl + "/" + api.createUserUrl, { body: user });
// ...
await api.post(api.baseUrl + "/posts/" + api.posts.create, post);
```

Модуль `api` не даёт чётких обещаний, как он будет работать, поэтому при его использовании нам надо знать, как он устроен. Но прямая зависимость от «внутренностей» повышает зацепление: если мы изменим модуль `api`, то придётся менять и код, который его использует. Это тормозит развитие приложения.

Вместо этого модуль `api` мог бы объявить *контракт* — набор гарантий, описывающих как он будет работать:

TYPESCRIPT

```
type ApiResponse = {
    state: "OK" | "ERROR";
};

interface ApiClient {
    createUser(user: User): Promise<ApiResponse>;
    createPost(post: Post): Promise<ApiResponse>;
}
```

Затем мы бы реализовали этот контракт внутри модуля `api`, не выпуская наружу лишних деталей:

```
const client: ApiClient = {
    createUser: (user) =>
        api.post(api.baseUrl + "/" + api.createUserUrl, { body: user }),

    createPost: (post) =>
        api.post(api.baseUrl + "/posts/" + api.posts.create, post),
};


```

...А после использовали бы его, уже опираясь только на контракт:

```
// ...
await client.createUser(user);
// ...
await client.createPost(post);
```

В формальном определении контракты — это пред- и постусловия в виде прописанных проверяемых спецификаций.[♂] На практике я почти не встречал контрактов в таком виде, зато в виде фиксирования гарантий — повсеместно.

Гарантии — это не обязательно сигнатура или интерфейс. Это могут быть устные или письменные договорённости, DTO, формат сообщений и т.д. Главное, чтобы эти договорённости *фиксировали поведение* частей системы друг перед другом.

Одни и те же обещания могут выполнять разные модули, поэтому если опираться на обещания, реализацию становится проще подменять, например, во время тестирования:

```
// Описываем «контракт» работы хранилища.  
// В интерфейсе указываем, какой метод можно использовать,  
// что он принимает в качестве аргумента  
// и что он возвращает как результат:  
  
interface SyncStorage {  
    save(value: string): void;  
}  
  
// В аргументе `saveToStorage` указываем не конкретную сущность,  
// а «что-то, что реализует интерфейс `SyncStorage`»:  
  
function saveToStorage(value: string, storage: SyncStorage) {  
    if (value) storage.save(value);  
}  
  
// ...  
  
describe("when given a non-empty value", () => {  
    // В тестах описываем мок хранилища,  
    // как «что-то, что реализует `SyncStorage`»:  
    const mock: SyncStorage = { save: jest.fn() };  
  
    it("should save it into the given storage", () => {  
        // Тогда во время тестов сможем «подменить»  
        // реализацию хранилища на мок:  
        saveToStorage("Hello World!", mock);  
        expect(mock.save).toHaveBeenCalled();  
    });  
});
```

...Или даже во время рантайма для замены одного алгоритма или куска приложения другим:

```
// Сохраняем настройки в куки или локальное хранилище
// в зависимости от пользовательских настроек:

const storage = preferences.useCookie ? cookieAdapter : localStorageAdapter;
const saveCurrentTheme = () => saveToStorage(THEME, storage);

// Пока `cookieAdapter` и `localStorageAdapter` оба реализуют `SyncStorage`,
// мы можем использовать любой из них в функции `saveToStorage`.
```

К СЛОВУ

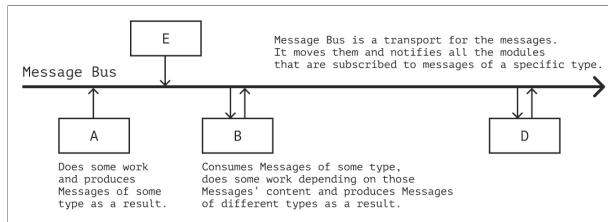
Идея такой «подмены» модулей лежит в основе шаблона «Стратегия» и инъекции зависимостей. ☺☺☺

События и сообщения

Чем ниже зацепление, тем больше общение модулей становится похоже на обмен сообщениями.

Общение между сервером и клиентом по REST — это пример такого общения. ☺ Клиент и сервер ничего не знают об устройстве друг друга и общаются только по заранее описанному контракту — сообщениями определённого вида с данными внутри.

Сообщения можно передавать как напрямую от одного модуля к другому через публичное API, так и через специальную сущность — *передатчик*. Во втором случае модули вообще ничего не будут знать друг о друге и будут зацеплены только через передатчика сообщений или событий:



Общение сводится к передаче и получению сообщений от передатчика

УТОЧНЕНИЕ

Я намеренно не называл передатчик «шиной событий», «очередью» или «брокером» сообщений. Между ними есть разница, ☺☺☺ но для сути этой главы она не критична, поэтому я не стал заострять внимание на конкретном термине.

Общение через события можно называть «идеальным общением» между модулями, потому что оно связывает модули только протоколами сообщений и их отправки. Но настройка такого общения часто ресурсозатратна, а для маленьких проектов и вовсе может оказаться оверхедом.

Передача событий и сообщений обычно ассоциируется с микросервисной архитектурой, но их преимущества можно использовать и в обычных приложениях. Если приложение большое, и надо построить общение между его частями без зацепления, то передатчик может помочь решить эту задачу.

В качестве примитивного передатчика можно представить паттерн «Наблюдатель». ⁹ В нём модули подписываются на сообщения определённых видов и реагируют на них, когда они приходят:

```
// Наблюдатель – функция, которая будет реагировать
// на сообщения `Message`:

type Observer = (message: Message) => void;

// Передатчик даёт возможность подписаться на события
// и может уведомить всех подписчиков о новом сообщении:

type Observable = {
    subscribe: (listener: Observer) => void;
    notifyAll: (message: Message) => void;
};

// Реализация передатчика в нашем случае – это объект с двумя методами
// и список подписчиков в виде массива `listeners`:

const listeners = [];
const bus: Observable = {
    subscribe: (listener) => listeners.push(listener),
    notifyAll: (message) => listeners.forEach((listener) => listener(message)),
};

// Подписчик будет знать, что ему на вход передадут сообщение `Message` – это кон-
// По типу сообщения он сможет понять, нужно ли ему обработать это сообщение:

const onUserUpdated = ({ type, payload }) => {
    if (type === "updateUser") {
        // Отреагировать на сообщение,
        // если тип подходит.

        const [firstName, lastName] = payload;
        // ...
    }
};

// Подписать функцию `onUserUpdated` на сообщения
// можно с помощью метода `subscribe`:

bus.subscribe(onUserUpdated);

// При появлении нового сообщения передатчик уведомит о нём
// всех подписавшихся с помощью метода `notifyAll`:
```

```
bus.notifyAll({  
    type: "updateUser",  
    payload: ["Alex", "Bespoyssov"],  
});
```

УТОЧНЕНИЕ

В продакшне писать свою реализацию «Наблюдателя», вероятно, не потребуется. Для работы с этим паттерном уже существуют решения, например, RxJS.[♂]

Полностью расцепленное общение нужно не всегда, но может быть полезно, когда *на одинаковые сообщения должны реагировать разные части системы*, но нам не хочется повышать зацепление между ними.

ЗАВИСИМОСТИ

Говоря о зацеплении и интеграции модулей, стоит упомянуть управление зависимостями. Под *зависимостями* для простоты будем иметь в виду любой код, который используется нашим.

Например, в функции `randomInt` кроме двух аргументов мы используем метод `Math.random` — это зависимость:

TYPESCRIPT

```
function randomInt(min, max) {  
    return Math.random() * (max - min) + min;  
}
```

К СЛОВУ

Зависимость от `Math` в примере выше *неявная*, потому что `Math` используется напрямую в теле функции и не обозначен, как аргумент. Такие неявные зависимости усиливают зацепление. Если попробовать протестировать функцию `randomInt`, нам придётся делать глобальный мок объекта `Math`.

Управлять зависимостями можно по-разному, это зависит (no pun intended) от парадигмы и стиля кода. Однако обычно удобно отделять зависимости, производящие эффекты, от остальных. Такое разделение помогает приводить код к *Impureim*-структуре, о пользе которой мы говорили в главе о сайд-эффектах. Далее мы посмотрим на примеры такого рефакторинга в коде, написанном в разных парадигмах.

Объектная композиция

В объектно-ориентированном программировании юнит композиции — это объект. Объекты могут смешивать данные и действия (состояние и методы), и поэтому компоновать объекты обычно труднее, чем функции. В частности большая часть паттернов проектирования и принципы SOLID решают именно проблемы объектной композиции. ☺

К СЛОВУ

Стоит отметить, что объектный код можно писать избегая этих проблем, если разделять данные и действия. Просто в функциональном программировании к этому подталкивает сама парадигма, а в ООП приходится прикладывать усилия, чтобы об этом помнить.

Для примера посмотрим на код приложения для управления финансами:

```

class BudgetManager {
    constructor(private settings: BudgetSettings, private budget: Budget) {}

    // Главная проблема кода в нарушении CQS: здесь эффекты смешаны с логикой.
    // Этот класс одновременно валидирует данные и обновляет значение бюджета...
    checkIncome(record: Record): MoneyAmount | boolean {
        if (record.createdAt > this.budget.endsAt) return false;

        const saving = record.amount * this.settings.piggyBankFraction;
        this.budget.topUp(record.amount - saving);

        return saving;
    }
}

// ...Но этого не видно на верхнем уровне композиции.
// Мы не сможем определить, что есть какой-то эффект,
// пока не посмотрим внутрь кода `BudgetManager`.
class AddIncomeCommandHandler {
    constructor(private manager: BudgetManager, private piggyBank: PiggyBank) {}

    execute({ record }: AddSpendingCommand) {
        const saving = this.manager.checkIncome(record);

        if (!saving) return false;
        this.piggyBank.add(saving);
    }
}

```

К СЛОВУ

В примере я подразумеваю, что мы используем *внедрение зависимостей* (*Dependency Injection, DI*) через конструкторы классов. Я не буду останавливаться на нём отдельно, но оставлю несколько ссылок, где об этом написано подробнее. ☺ ☺

В примере выше из-за нарушения CQS нам не ясно, сколько эффектов на самом деле происходит при вызове метода `execute`. Мы увидели 2, но нет гарантий, что `this.budget.topUp` не меняет что-нибудь кроме объекта `budget`.

Компоновка сайд-эффектов сводит на нет суть абстракции: чем больше эффектов, тем больше общего состояния, за которым надо следить и держать в голове — с этим трудно работать. Если компоновки

сайд-эффектов можно избежать, то лучше её избежать.

Вместо этого можно выделить преобразования данных, а эффекты отодвинуть к краям приложения.

Так получилось бы разделить эффекты и логику:

```

// Валидацию вынесем в отдельную сущность.
// Этот класс будет заниматься только валидацией.

class AddIncomeValidator {
    constructor(private budget: Budget) {}

    // При необходимости, метод `canAddIncome` можно сделать полностью чистым,
    // если передавать значение `endsAt`, как аргумент.
    canAddIncome(record: Record) {
        return record.createdAt <= this.budget.endsAt;
    }
}

// На верхнем уровне разделим логику и эффекты.

// Получится Impure-im-бутерброд, о котором мы говорили раньше:
// - Эффекты для получения данных (например, работа с БД на бекенде);
// - Логика обработки данных (доменные функции, создание сущностей);
// - Эффекты для сохранения данных (или вывода на экран).

class AddIncomeCommandHandler {
    constructor(
        // Эта же техника позволит «вытащить» наружу проблемы с зацеплением.
        // Если в классе набирается слишком много зависимостей,
        // то, вероятно, стоит подумать об улучшении его дизайна.
        private settings: BudgetSettings,
        private validator: AddIncomeValidator,
        private budget: Budget,
        private piggyBank: PiggyBank
    ) {}

    execute({ record }: AddSpendingCommand) {
        // Валидация:
        if (!this.validator.validate(record)) return false;

        // Pure(-ish because of injected settings) logic,
        // can be extracted into a separate module:

        // Чистая (почти – из-за внедрённых `this.settings`) логика.
        // Её можно вынести в отдельный метод или модуль,
        // я решил оставить её здесь для наглядности «слойёв бутерброда»:
        const saving = record.amount * this.settings.piggyBankFraction;
        const income = record.amount - saving;

        // Эффекты для сохранения данных:
    }
}

```

```
        this.budget.topUp(income);
        this.piggyBank.add(saving);
    }
}
```

Разделение данных и поведения

Следующим шагом было бы хорошо отделить данные от поведения. Объекты бюджета и копилки тогда превратились бы в «контейнеры данных» — *сущности (entities в терминах DDD)*, а сохранением данных занимались бы «сервисы»:

TYPESCRIPT

```
class AddIncomeCommandHandler {
    constructor(
        private settings: BudgetSettings,
        private validator: AddIncomeValidator,
        private budgetRepository: BudgetUpdater,
        private piggyBankRepository: PiggyBankUpdater
    ) {}

    // Объекты бюджета и копилки теперь не содержат поведения, только данные:
    execute({ record, budget, piggyBank }: AddSpendingCommand) {
        if (!this.validator.validate(record, budget)) return false;

        // Преобразования данных, бизнес-логика:
        const saving = record.amount * this.settings.piggyBankFraction;
        const income = record.amount - saving;

        // Обновлённые объекты с данными:
        const newBudget = new Budget({ ...budget, income });
        const newPiggyBank = new PiggyBank({ ...piggyBank, saving });

        // «Сервисы» с эффектами сохранения данных:
        this.budgetRepository.update(newBudget);
        this.piggyBankRepository.update(newPiggyBank);
    }
}
```

Разделение данных и поведения помогает отделять код, который меняется быстро (поведение), от кода, который меняется медленно (данные). Так изменения затрагивают меньше файлов, и это ограничивает их распространение по кодовой базе.

Принцип разделения интерфейса

При описании поведения мы можем использовать CQS,⁹ как «интеграционный линтер». Например, если мы описываем сервис с набором команд, разница сигнатур может указать на нарушение CQS:

TYPESCRIPT

```
interface BudgetUpdater {  
    updateBalance(balance: MoneyAmount): void;  
    recalculateDuration(date: TimeStamp): void;  
  
    // Упс! Метод что-то возвращает,  
    // значит это запрос, а не команда:  
    currentBalance(): MoneyAmount;  
}
```

Тогда мы можем применить *принцип разделения интерфейса* (*Interface Segregation Principle, ISP*)⁹ и декомпозировать задачу:

TYPESCRIPT

```
interface BudgetSource {  
    currentBalance(): MoneyAmount;  
}  
  
interface BudgetUpdater {  
    updateBalance(balance: MoneyAmount): void;  
    recalculateDuration(date: TimeStamp): void;  
}
```

К СЛОВУ

Один из паттернов применения ISP — разделение сервисов чтения и записи данных. Разделять ли сервисы на уровне классов — зависит от задачи, но разделение на уровне интерфейсов помогает как минимум выразить разницу намерений.

Функциональная композиция

В проектах с «более функциональным кодом» тоже можно встретить «внедрение зависимостей», сделанное с помощью частичного применения функций. Польза такого внедрения в том, что оно делает *неявные зависимости явными*.

Например, функция `listingQuery` выводит список md-файлов из указанной папки:

```

const listingQuery = (query) => {
  return fs
    .readdirSync(query)
    .filter((fileName) => fileName.endsWith(".md"))
    .map((fileName) => fileName.replace(".md", ""));
};

const projectList = listingQuery("projects");

```

Она неявно зависит от модуля `fs`, который предоставляет доступ к файловой системе. В целом, это не страшно, но такую функцию неудобно тестировать — для её тестов потребуется глобальный мок для `fs`.

С частичным применением можно создать «функцию-фабрику». Она будет принимать «зависимости», как аргумент, и возвращать функцию `listingQuery`, как результат:

```

/**
 * 1. Внедрение «сервиса» system;
 * 2. Передача собственно аргументов;
 * 3. Использование «сервиса» system
 *     для получения нужных эффектов.
 */

const createListingQuery =
  ({ system }) =>
  (query) =>
    system
      .readdirSync(query)
      .filter((fileName) => fileName.endsWith(".mdx"))
      .map((fileName) => fileName.replace(".mdx", ""));

// Тогда при использовании мы бы сперва «внедрили» system:
const listingQuery = createListingQuery({ system: fs });

// ...А затем бы использовали созданную функцию:
const projectList = listingQuery("projects");

```

Этот подход «не очень функциональный», но с ним вполне можно работать, если нам не трудно «внедрять» зависимости для каждой такой фабрики, а их использование не доставляет проблем с общим состоянием или эффектами.

Отказ от зависимостей

В «более хардкорном» ФП менять состояние и производить сайд-эффекты не принято. Понятие «зависимостей» в привычном понимании там не очень подходит. Вместо «зависимостей», которые «надо дёргать за методы», и эффектов ФП предлагает функциональное ядро в императивной оболочке.

К слову

Мне нравится, как эту концепцию Марк Симанн называет — отказ от зависимостей. ↗ Мы как бы отходим от концепции зависимостей вообще и пробуем решить проблему иначе.

В этом подходе вся работа с состоянием отодвинута к краям приложения. Это значит, что читать и записывать (или выводить на экран) данные можно только в начале работы модуля и в конце. Вся работа между этими точками строится на преобразованиях данных.

То есть мы сперва получаем все нужные данные из «грязного» мира, передаём их как аргументы в цепочку преобразований, а потом записываем результат:

```

// Выносим функцию с преобразованием
// названий файлов к названиям постов,
// список которых надо вернуть:
const listingQuery = (fileNames) =>
  // Заметьте: нет зависимости от `fs`,
  // мы работаем только с данными:
  fileNames
    .filter((fileName) => fileName.endsWith(".mdx"))
    .map((fileName) => fileName.replace(".mdx", ""));

// «Композиция» теперь – это отдельная функция:
// - она сперва вызывает нужные эффекты, чтобы получить данные,
// - потом прогоняет их через цепочку вычислений,
// - а в конце возвращает результат или вызывает эффект для сохранения данных:
const listingQueryComposition = (query) => {
  // Внутри используем Imperative-бутерброд.
  //
  // 1. Эффект для получения данных.
  //     (Именно из-за присутствия эффектов контекст композиции
  //     и саму функцию композиции считают «нечистыми».)
  const files = fs.readdirSync(query);

  // 2. Логика преобразований в виде цепочки чистых функций.
  return listingQuery(files);

  // 3. Эффект для сохранения данных.
  //     (Если мы не возвращаем результат из функции,
  //     а сохраняем его, то после завершения цепочки преобразований
  //     мы бы вызвали эффект сохранения данных.)
};


```

С первого взгляда кажется, что стало хуже: для тестов понадобятся глобальные моки, а с «внедрением» можно использовать заглушки на место сервисов. Но в этой концепции юнит-тестами надо тестиировать только функциональное ядро — функцию `listingQuery`. Композицию в простых случаях можно вообще не тестиировать, а в более сложных — использовать интеграционные или E2E-тесты.

При использовании же интеграционных тестов такая композиция подтолкнёт нас к архитектуре «Порты-адаптеры», которая поможет уменьшить количество моков, что сделает тесты менее «хрупкими». ☺

Об архитектуре «Порты-адаптеры» мы ещё поговорим в отдельной главе.

Моки нам всё ещё понадобятся для, например, тестирования адаптеров. Но в этом случае на каждый сервис мы напишем лишь один адаптер, а значит и мокать этот сервис нужно будет лишь один раз.

TYPESCRIPT

```
interface System {
    readDirectory(directory: DirectoryPath): List<FileName>;
}

const createAdapter = (service): System => ({
    readDirectory(directory) {
        /*...*/
    },
});

// test.js

describe("when asked to read the given directory", () => {
    it("should trigger the `readdirSync` method on the service", () => {
        const mock = { readdirSync: jest.fn() };
        const adapter = createAdapter(mock);

        adapter.readDirectory("testdir");
        expect(mock.readdirSync).toHaveBeenCalledWith("testdir");
    });
});

```

В случае с «внедрением зависимостей» мокать сервис приходилось бы для каждой функции, куда этот сервис внедрён.

Другие способы управления зависимостями

Если в проекте очень много операций ввода-вывода, то тогда «внедрение зависимостей» через частичное применение подойдёт, вероятно, лучше.

Тем не менее, даже если мы собираемся «внедрять» сервисы, это будет сделать гораздо проще, если первым делом мы разделим логику и эффекты. Поэтому в функциональном коде разделение логики и эффектов — это первый рефакторинг, который стоит сделать.

В ФП используют и другие техники работы с зависимостями. ↪ ↪ Я в продакшен-коде на JavaScript такое видел только один раз, поэтому не могу сказать, насколько они удобны.

Хороший обзор техник работы с «зависимостями» в ФП описал Скотт Влашин в своём цикле статей “Six approaches to dependency injection”. ↪ Очень рекомендую по крайней мере первые три статьи.

Целостность и согласованность

В приложениях с состоянием нам также стоит следить за *согласованностью* и *целостностью* данных. Это свойства, которые обеспечивают непротиворечивость того, что видит пользователь.

[ПОДРОБНЕЕ](#)

Подробно об этом написал Скотт Влашин в “Domain Modeling Made Functional” в разделе об агрегатах и согласованности данных. ↪

Агрегаты

Согласованность обеспечить проще всего, если использовать неизменяемые структуры данных и следить за инкапсуляцией.

Рассмотрим на примере. Представим, что в коде интернет-магазина корзина должна всегда иметь правильную итоговую сумму:

```
// cart.ts
type Cart = {
    items: ProductList;
    total: MoneyAmount;
};

function totalPrice(products: ProductList) {
    return products.reduce(
        (tally, { price, count }) => tally + price * amount,
        0
    );
}

function createCart(products: productList): Cart {
    return {
        items: products,
        total: totalPrice(products),
    };
}
```

Неправильно изменив объект корзины, мы можем рассогласовать данные. Допустим, какой-то посторонний код добавил новый продукт в список:

```
// Посторонний код не знает, что после добавления продукта
// надо ещё и пересчитать итоговую цену:
userCart.products.push(appleJuice);

// Теперь поле `cart.total` показывает неправильное значение,
// потому что после добавления продукта оно не было пересчитано.
```

Кусок данных, которые должны обновляться «как единое целое» — это *агрегат*. Неизменяемость данных может помочь держать агрегаты согласованными. Она принуждает к тому, чтобы обновление агрегата происходило, начиная с его корневого уровня:

```
// cart.ts  
// ...  
  
function addProduct(cart: Cart, product: Product): Cart {  
    const products = [...cart.products, product];  
    const total = totalPrice(products);  
    return { products, total };  
}  
  
// ...  
  
addProduct(userCart, appleJuice);
```

Функция `addProduct` гарантирует согласованность, потому что она знает, какие данные и откуда обновлять, чтобы они были валидными. Правильное обновление — её зона ответственности, агрегат — область влияния.

Предвалидация на входе в контекст

Агрегаты и неизменяемость помогают держать данные *внутри* части приложения валидными и согласованными. А чтобы невалидные данные в него *не попали*, нужна предвалидация.

[ПОДРОБНЕЕ](#)

В терминах DDD обособленная часть приложения называется ограниченным контекстом.

Подробнее об этом понятии мы говорили ранее в главе о функциональном пайплайне.

То есть в компоненте `Cart` вместо ад-хок проверок на нужные поля:

```

function Cart({ items }) {
  return (
    !!items && (
      <ul>
        {items.map((item) =>
          item && item.product ? (
            <li key={item.id}>
              {item.product?.name ?? "-": item.product?.price} ×{" "}
              {item.product?.count ?? 0}
            </li>
          ) : null
        )}
      </ul>
    )
  );
}

```

...Можно сперва провалидировать данные, обработать потенциальные ошибки:

```

function hasCorruptedItem(item) {
  return !item || !item.product;
}

function validateCart(cart) {
  if (!cart || !cart.items) return Result.failure("EMPTY_CART");
  if (cart.items.some(hasCorruptedItem))
    return Result.failure("CORRUPTED_ITEM");

  // Вместо Result ваш проект может использовать исключения.
  // Подробнее об обработке ошибок мы говорили в предыдущей главе.

  // Альтернативой ошибкам может быть «дефолтная» корзина как результат валидации
  // но мне всё же кажется, что нарушение контракта API – достаточная причина для

  return cart;
}

```

...А внутри компонента использовать уже *валидные и проверенные* данные:

```
// ...  
  
function Cart({ items }) {  
  return (  
    <ul>  
      {items.map(({ id, product }) => (  
        <li key={id}>  
          {product.name}: {product.price} × {product.count}  
        </li>  
      ))}  
    </ul>  
  );  
}
```

-
1. Засцепление в программировании, Википедия, [https://ru.wikipedia.org/wiki/Засцепление_\(программирование\)](https://ru.wikipedia.org/wiki/Засцепление_(программирование))
 2. Разделение ответственности, Википедия, https://ru.wikipedia.org/wiki/Разделение_ответственности
 3. Связность в программировании, Википедия, [https://ru.wikipedia.org/wiki/Связность_\(программирование\)](https://ru.wikipedia.org/wiki/Связность_(программирование))
 4. Feature Envy, Refactoring Guru, <https://refactoring.guru/smells/feature-envy>
 5. "Design By Contract", c2.com, <https://wiki.c2.com/?DesignByContract>
 6. «Контрактное программирование» Тимур Шемединов, https://youtu.be/K5_lSuVbGEO.
 7. Strategy Pattern, Refactoring Guru, <https://refactoring.guru/design-patterns/strategy>.
 8. "Inversion of Control Containers and the Dependency Injection pattern" by Martin Fowler, <https://martinfowler.com/articles/injection.html>
 9. Внедрение зависимостей с TypeScript на практике, <https://bespoyasov.ru/blog/di-ts-in-practice/>
 10. REpresentational State Transfer, REST, <https://restfulapi.net>
 11. Message Broker, Microsoft Docs, [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10)).
 12. Message Bus, Microsoft Docs, [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10)).
 13. Message Queue, Wikipedia, https://en.wikipedia.org/wiki/Message_queue
 14. Observer Pattern, Refactoring Guru, <https://refactoring.guru/design-patterns/observer>
 15. RxJS, Reactive Extensions Library for JavaScript, <https://rxjs.dev>

16. The Principles of OOD, Robert C. Martin, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
17. "Dependency Injection in .NET" by Mark Seemann, <https://www.goodreads.com/book/show/9407722-dependency-injection-in-net>
18. "Domain-Driven Design" by Eric Evans,
https://www.goodreads.com/book/show/179133.Domain_Driven_Design
19. "Evans Classification" by Martin Fowler, <https://martinfowler.com/bliki/EvansClassification.html>
20. "Functional architecture: The pits of success" by Mark Seemann, <https://youtu.be/US8QG9l1XW0>
21. "Command-Query Separation" by Martin Fowler,
<https://martinfowler.com/bliki/CommandQuerySeparation.html>
22. "Dependency Rejection" by Mark Seemann, <https://blog.ploeh.dk/2017/02/02/dependency-rejection/>
23. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov,
<https://www.goodreads.com/book/show/48927138-unit-testing>
24. "Six approaches to dependency injection" by Scott Wlaschin,
<https://fsharpforfunandprofit.com/posts/dependencies/>
25. "Dependency Injection Using the Reader Monad" by Scott Wlaschin,
<https://fsharpforfunandprofit.com/posts/dependencies-3/>
26. "Dependency Interpretation" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/dependencies-4/>
27. "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>

Обобщения и иерархии

Во время рефакторинга мы можем обнаружить в коде «скрытые закономерности» в разных частях приложения. Это могут быть схожие алгоритмы или принципы работы функций, которые можно обобщить.

Обобщения могут как упростить код, так и сделать его сложнее. В этой главе мы поговорим о том, как понять, когда стоит заменять похожий код обобщёнными алгоритмами и типами, а когда — нет. Мы также затронем тему иерархий и наследования и поговорим, почему композиция предпочтительнее наследования.

Обобщённые алгоритмы

Обобщённый алгоритм — это продолжение идей абстракции и уменьшения дублирования кода.

Если несколько функций работают «по одинаковой схеме», мы можем оформить эту «схему» в виде набора операций. Такая схема будет описывать работу функций абстрактно, «в общих чертах», не ссылаясь на конкретные переменные в коде.

Например, вместо ручного перебора элементов массива:

```

const items = [1, 2, 3, 4, 5];

for (const i of items) {
  console.log(i);
}

// 1 2 3 4 5

for (const i of items) {
  console.log(i - 1);
}

// 0 1 2 3 4

```

...Мы можем выделить «схему работы», а именно — перебрать массив, для каждого элемента применить указанную функцию. В примере ниже эту схему инкапсулирует в себе метод `forEach`:

```

const printNumber = (i) => console.log(i);
const printNumberMinusOne = (i) => console.log(i - 1);

// Перебрать массив `items`, для каждого элемента выполнить функцию `printNumber`
items.forEach(printNumber);
// 1 2 3 4 5

// Перебрать массив `items`, для каждого элемента выполнить функцию `printNumberMinusOne`
items.forEach(printNumberMinusOne);
// 0 1 2 3 4

```

Обобщённый алгоритм «схемы» не ссылается на конкретные переменные и функции. Вместо этого он ожидает их как параметры:

Перебрать указанный массив; для каждого элемента применить указанную функцию



Такой алгоритм опирается на *признаки* переданных параметров. Массив можно перебрать по-элементно, функцию можно вызвать, передав аргументом элемент массива. Сочетание этих признаков — это контракт на работу такого алгоритма. Выявление подобных схем и определение их контрактов — и есть основа обобщённого программирования.

Чтобы понять, чем обобщённые алгоритмы могут быть полезны и как их выявлять, посмотрим на приложение для управления финансами. В коде ниже есть две функции для подсчёта трат и пополнений в истории:

```
// Тип описывает историю трат и доходов в приложении:  
type RecordHistory = List<Entry>;  
  
// Запись в истории содержит вид, дату создания и сумму:  
type EntryKind = "Spend" | "Income";  
type Entry = {  
    type: EntryKind;  
    created: TimeStamp;  
    amount: MoneyAmount;  
};  
  
// Подсчитывает, сколько всего потрачено:  
function calculateTotalSpent(history: RecordHistory) {  
    return history.reduce(  
        (total, { type, amount }) => total + (type === "Spend" ? amount : 0),  
        0  
    );  
}  
  
// Подсчитывает, сколько всего добавлено:  
function calculateTotalAdded(history: RecordHistory) {  
    let total = 0;  
    for (const { type, amount } of history) {  
        if (type === "Income") total += amount;  
    }  
    return total;  
}
```

Допустим, в приложение понадобилось также добавить подсчёт трат за некоторый период, например, за сегодня. При добавлении нам стоит проверить, нет ли в новой функции общих черт с уже имеющимися:

```
// В новой функции `calculateSpentToday` видна «схема работы»,
// похожая на алгоритм из `calculateTotalSpent` и `calculateTotalAdded`:
// - взять массив истории;
// - найти нужные записи;
// - просуммировать.

function calculateSpentToday(history: RecordHistory) {
    return history.reduce(
        (total, { type, created }) =>
        total +
        (type === "Spend" && created >= today && created < tomorrow ? amount : 0),
        0
    );
}
```

Заметим, что каждая из трёх функций сводится к двум задачам: отфильтровать историю и просуммировать поле `amount` в отфильтрованных записях. Эти задачи мы можем вынести наружу и обобщить:

```
type HistorySegment = List<Entry>;
type EntryPredicate = (record: Entry) => boolean;

// Фильтрация теперь может использоваться отдельно.
// При этом функция `keepOnly` может фильтровать историю записей
// по _любому_ критерию, который реализует тип `EntryPredicate`:

const keepOnly = (
    history: RecordHistory,
    criterion: EntryPredicate
): HistorySegment => history.filter(criterion);

// Суммирование теперь тоже может использоваться отдельно.
// При этом в функцию `totalOf` можно передать _любой_ фрагмент истории,
// и она посчитает сумму трат или доходов в ней:

const totalOf = (history: HistorySegment): MoneyAmount =>
    history.reduce((total, { amount }) => total + amount);
```

Тогда предыдущие функции мы сможем скомпоновать из этих двух задач — это и будет реализацией обобщённого алгоритма. Отличающиеся детали мы будем использовать как *параметры* этого

обобщённого алгоритма:

TYPESCRIPT

```
// Отличаются лишь критерии фильтрации, всё остальное одинаково:  
const isIncome: EntryPredicate = ({ type }) => type === "Income";  
const isSpend: EntryPredicate = ({ type }) => type === "Spend";  
const madeToday: EntryPredicate = ({ created }) =>  
    created >= today && created < tomorrow;  
  
// Критерий фильтрации передаём как «параметр» для `keepOnly`:  
const added = keepOnly(history, isIncome);  
const spent = keepOnly(history, isSpend);  
  
// Подсчёт суммы тогда можно вести по любому фрагменту истории:  
totalOf(spent);  
totalOf(added);  
totalOf(keepOnly(spent, madeToday));
```

Обобщения удобны, когда мы полностью уверены в «схеме работы». Если у нескольких кусков кода «схема» одинаковая или отличается незначительно, обобщение может помочь уменьшить дублирование.

Но обобщения также могут сделать код сложнее. Если мы подозреваем, что «схема работы» может поменяться, то обобщать рано. Эвристика здесь примерно та же, что и при работе с дублированием. Пока мы не уверены в своих знаниях о коде, с обобщением лучше повременить.

К СЛОВУ

Правила работы с дублированием, а также то, как отличать повторяющийся код от недостатка информации, мы подробнее обсуждали в одной из предыдущих глав.

Обобщённые типы

Иногда схожие детали могут быть не у алгоритмов, а у типов данных. *Tip* — это набор значений с некоторыми определёнными свойствами. ↝ ↝ Похожие типы можно объединить в обобщённый тип, но при этом стоит следовать правилу:

Обобщать только когда есть уверенность, что не будет исключений

Данные менять сложнее, чем код. Неаккуратное обобщение типов может ослабить типизацию и вынудить добавлять в код лишние проверки. Для примера вернёмся к типу `Entry` из фрагмента выше:

TYPESCRIPT

```
type EntryKind = "Spend" | "Income";
type Entry = {
    type: EntryKind;
    created: TimeStamp;
    amount: MoneyAmount;
};
```

Если мы знаем, что в истории могут лежать *только* записи с такими свойствами, то тип хорошо описывает предметную область. Но если есть вероятность, что в истории могут находиться и другие записи, то с этим типом могут возникнуть проблемы.

Допустим, в истории могут отображаться комментарии пользователя, которые не содержат суммы, но содержат текст:

TYPESCRIPT

```
type EntryKind = "Spend" | "Income" | "Comment";
type Entry = {
    type: EntryKind;
    created: TimeStamp;

    // Обобщение ослабляет поле `amount`,
    // а вместе с ним и весь тип `Entry`:
    amount?: MoneyAmount;

    // При этом появляется новое поле, тоже слабое –
    // «может быть, а может не быть»:
    content?: TextContent;
};
```

В типе `Entry` есть противоречие. Его поля `amount` и `content` — необязательные, поэтому тип «разрешает» записи без суммы и текста. Но это неправильное отражение предметной области: комментарий должен содержать текст, а траты и пополнения должны содержать сумму. Без этого условия данные в истории записей будут невалидными.

Проблема в том, что тип `Entry` с необязательными полями пытается смешать в себе *разные* сущности и состояния данных. Мы можем проверить это, если создадим React-компонент для вывода суммы записи из истории на экран:

```
type RecordProps = {
  record: Entry;
};

// В компонент передаём запись типа `Entry`,
// чтобы вывести сумму траты или пополнения:
const Record = ({ record }: RecordProps) => {
  // Но внутри нам приходится фильтровать эти данные.
  // Если запись — это комментарий, рендер придётся пропустить,
  // комментарии суммы не содержат:
  if (record.type === "Comment") return null;

  const sign = isIncome(record) ? "+" : "-";

  // А тут придётся ещё и сказать компилятору,
  // что `amount` «точно есть, мы проверили!»
  return `${sign} ${record.amount!}`;
};
```

Рендер комментария в этом компоненте не особо приживается. Чтобы исправить эту проблему, нам сперва нужно понять, что мы знаем о предметной области:

- Здесь могут быть только комментарии, или могут появиться другие текстовые сообщения?
- Появится ли новый тип записей, в котором будет поле `amount`?
- Могут ли быть новые типы записей, в которых будут совершенно другие поля?

На такие вопросы ответить можно не всегда. Когда нам не хватает знаний о предметной области или требованиях к программе, обобщать типы рано. В таких случаях предпочтительнее вместо обобщений типы компоновать:

```

// Мы разбили тип на несколько.
// Необязательные поля пропали, мы чётче разделяем состояния данных,
// а от статической типизации будет больше толку.

type Spend = { type: "Spend"; created: TimeStamp; amount: MoneyAmount };
type Income = { type: "Income"; created: TimeStamp; amount: MoneyAmount };
type Comment = { type: "Comment"; created: TimeStamp; content: TextContent };

// Да, мы написали «больше кода», но на ранних стадиях проекта
// нам важнее попасть в предметную область, чтобы правильно понять
// суть приложения и связи между сущностями.

// Такие атомарные типы, как выше, — гибче,
// потому что мы можем компоновать их по признакам,
// которые нам важны в конкретной ситуации.

// Например, ниже в типе `FinanceEntry`
// собираем только записи, _содержащие сумму_.
// По спискам таких записей можно проходить сумматором:

type FinanceEntry = Spend | Income;

// В типе `MessageEntry` собираем записи, _содержащие текст_.
// Сейчас такой тип только один, поэтому `MessageEntry` — это алиас,
// но если записей с текстом станет больше, мы сможем расширить этот тип дальше.

type MessageEntry = Comment;

// Самый общий тип `Entry` представим как выбор из _всех_ возможных вариантов.
// С такими записями можно делать только то, что можно делать с _любой_ записью:
type Entry = FinanceEntry | MessageEntry;

// Например, `Entry` можно сортировать по дате появления,
// так как `created` гарантированно есть у всех записей:
const sortByDate = (a: Entry, b: Entry) => a.created - b.created;

```

После рефакторинга компоненту `Record` больше не нужны лишние проверки:

```
// В пропсах указываем не общий `Entry`, а `FinanceEntry`.  
// В нём по определению есть сумма, поэтому дополнительных  
// проверок во время рендерна нам не потребуется.  
  
type RecordProps = {  
    record: FinanceEntry;  
};  
  
const Record = ({ record }: RecordProps) => {  
    const sign = isIncome(record) ? "+" : "-";  
    return `${sign} ${record.amount}`;  
};
```

К СЛОВУ

Даже если мы сделаем скидку на «ненастоящую типизацию» в TypeScript и ошибки рантайма, то одно лишь проектирование с оглядкой на преждевременные обобщения избавят код от лишних проверок.

Конечно, нет гарантий, что в продакшене в пропс компонента не попадёт неправильный тип данных. Но с грамотным алёрт-мониторингом и обработкой ошибок мы сможем это быстро найти и исправить.

Основная идея компоновки типов в том, чтобы не обобщать *раньше времени*. Скомпонованные типы проще расширять по мере усложнения требований к приложению. К примеру, нам потребовалось добавить записи с овердрафтом и системные текстовые сообщения. Добавим `Overdraft` и `Warning`:

```

type Spend = { type: "Spend"; created: TimeStamp; amount: MoneyAmount };
type Income = { type: "Income"; created: TimeStamp; amount: MoneyAmount };
type Overdraft = { type: "Overdraft"; created: TimeStamp; amount: MoneyAmount };

type Comment = { type: "Comment"; created: TimeStamp; content: TextContent };
type Warning = { type: "Warning"; created: TimeStamp; content: TextContent }; //

// В юнионах нам достаточно добавить по ещё одному варианту
// в каждое место, которое требуется расширить:
type FinanceEntry = Spend | Income | Overdraft;
type MessageEntry = Comment | Warning;
type Entry = FinanceEntry | MessageEntry;

// При необходимости мы можем перекомпоновать типы с нуля,
// чтобы скомпоновать их по другим признакам.

```

А вот когда мы знаем о предметной области достаточно, можно при желании выделить закономерности и обобщить типы. (Мы знаем достаточно, когда код этих сущностей перестал меняться.) Но это совсем необязательный шаг:

```

type FinanceEntry = {
    type: "Spend" | "Income" | "Overdraft";
    created: TimeStamp;
    amount: MoneyAmount;
};

type MessageEntry = {
    type: "Comment" | "Warning";
    created: TimeStamp;
    content: TextContent;
};

```

Наследование и композиция

В примере выше для описания типа `Entry` возникает соблазн использовать какой-нибудь дженерик-тип \mathcal{X} , чтобы определять вид записи «на лету»:

```

type FinanceEntryKind = "Spend" | "Income" | "Overdraft";
type MessageEntryKind = "Comment" | "Warning";
type EntryKind = FinanceEntryKind | MessageEntryKind;

// Поля и поведение тогда будут определяться
// с помощью тип-аргумента `TKind`:

type Entry<TKind extends EntryKind> = {
    type: TKind;
    created: TimeStamp;
    amount: TKind extends FinanceEntry ? MoneyAmount : never;
    content: TKind extends MessageEntryKind ? TextContent : never;
};

type Income = Entry<"Income">;
type Comment = Entry<"Comment">;

```

Но с дженериками тоже лучше не торопиться. Дженерики — как чертежи для типов. Чтобы использовать их, нам надо быть уверенными, что структура типа не поменяется.

ОСТОРОЖНО

Этот раздел — одно большое IMHO, могу быть абсолютно неправ, настройте скептицизмометры на максимум.

Дженерики подойдут для случаев когда мы знаем структуру типа, но не знаем его деталей. Например, в коде выше у нас был тип `EntryPredicate`:

```
type EntryPredicate = (record: Entry) => boolean;
```

Предикат — это *по определению* функция, которая возвращает булево значение, поэтому мы точно знаем структуру:

```
type SomethingPredicate = (x: Something) => boolean;
```

Если нам по какой-то причине потребуется создать много предикатов от *разных аргументов*, но заранее неизвестно каких, то дженерик отлично опишет такой «чертёж»:

```
// Предикат от «абстрактного параметра»:
type Predicate<T> = (x: T) => boolean;

// Предикаты от конкретных параметров:
type EntryPredicate = Predicate<Entry>;
type HistoryPredicate = Predicate<History>;
type WhateverPredicate = Predicate<Whatever>;
```

Здесь мы уверены, что *структура типа известна и не изменится*. Мы можем передать любой тип-аргумент, и это не отразится на структуре типа и его работе. В `Entry<TKind>` такой гарантии дать нельзя, по крайней мере на ранних этапах проектирования.

Наследование

Раз мы заговорили о компоновке и «чертежах» сущностей, затронем ООП и наследование. В проектах, написанных в ООП-стиле, лучше избегать глубоких иерархий наследования:

```
class Task {
    public start(): void {}
}

class AdvancedTask extends Task {
    public configure(settings: TaskSettings): void {}
}

class UserTask extends AdvancedTask {
    public definedBy: User;
}

// Класс `UserTask` 3-й в цепочке наследования:
// Task -> AdvancedTask -> UserTask
```

Глубокие иерархии — хрупкие. Они претендуют на точное знание предметной области, но модель не может описать мир *точно*, поэтому рано или поздно иерархия поломается. Если мы не предусмотрели какую-то функциональность, то из-за сломанной иерархии придётся либо добавлять эту функциональность в базовый класс, либо переопределять наследников, ломая принцип подстановки (об этом чуть позже).

Вместо наследования предпочтительнее использовать композицию. В ООП-коде композиция обычно означает реализацию интерфейсов:

```
// Объявим несколько интерфейсов,
// каждый из которых описывает связанный набор фич:

interface SimpleTask {
    start(): void;
}

interface Configurable {
    configure<TSettings>(settings: TSettings): void;
}

interface UserDefined {
    definedBy: User;
}

// Класс может реализовать несколько интерфейсов,
// тем самым минуя ненужные шаги с наследованием:

class UserTask implements SimpleTask, Configurable, UserDefined {}

// Тогда при добавлении новой фичи, нам будет достаточно
// расширить список интерфейсов новым и реализовать его:

interface Cancellable {
    stop(): void;
}

class UserTask
    implements SimpleTask, Restartable, Configurable, UserDefined, Cancellable {}

// В случае с наследованием нам бы пришлось
// менять один из базовых классов
// или менять структуру наследования.
```

ОДНАКО

Мы здесь не говорим о реализации абстрактных классов, она наоборот бывает полезна. Абстрактный класс — это почти что «интерфейс с дефолтным поведением», поэтому «это другое». Но даже с абстрактными классами лучше избегать иерархий глубже 1–2 уровней.

В JavaScript, однако, есть один юзкейс, когда глубокое наследование может быть полезным. Если в проекте для обработки ошибок используются паники, но хочется избежать многословных проверок с `instanceof` на каждый тип ошибок, то можно использовать иерархии «слоёв ошибок»:

TYPESCRIPT

```
// Базовый класс всех ошибок приложения:  
class AppError extends Error {}  
  
// Классы ошибок, разделённые по «слоям»:  
// Слой API и его подклассы на каждую ошибку API.  
class ApiError extends AppError {}  
class NotFound extends ApiError {}  
class BadRequest extends ApiError {}  
  
// Слой валидации и подклассы на каждую ошибку валидации.  
class ValidationError extends AppError {}  
class InvalidUserDto extends ValidationError {}  
class MissingPostData extends ValidationError {}  
  
// Тогда при обработке мы можем сократить количество проверок,  
// заменив проверки на каждый возможный тип:  
if (e instanceof NotFound) {  
} else if (e instanceof BadRequest) {  
} else if (e instanceof InvalidUserDto) {  
} else if (e instanceof MissingPostData) {  
}  
  
// ...На проверки «по слоям» ошибок:  
if (e instanceof ApiError) {  
} else if (e instanceof ValidationError) {  
}
```

Так мы можем уменьшить количество проверок с `instanceof`, но это будет работать, только если для всех ошибок из одного слоя обработка будет одинаковой.

Принцип подстановки Лисков

Чуть выше мы приводили пример с компонентом, в котором были лишние проверки:

```

type Entry = Spend | Income | Comment;
type RecordProps = { record: Entry };

const Record = ({ record }: RecordProps) => {
  if (record.type === "Comment") return null; // Фильтр комментариев.

  const sign = isIncome(record) ? "+" : "-";
  return `${sign} ${record.amount}`;
};

```

Когда мы видим подобные условия, нам стоит проверить, не нарушен ли принцип подстановки Лисков. ☹️☹️ Этот принцип в прикладной формулировке Мартина звучит так:

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом ☹️ //

На практике это значит, что тип `Entry` описывает любую запись: не важно, комментарий, трату или доход. Поэтому и передать его можно только в ту функцию, которая *готова работать с любой записью*.

То есть если функция собирается работать только с тратой или доходом, но не комментарием, то тип `Entry` в такую функцию передавать нельзя. Нельзя потому, что функция может захотеть достать из записи количество денег, но в комментарии его нет, поэтому его надо отсеять перед использованием.

И наоборот, если функция готова работать с любой записью и будет использовать, например, только поле `created`, то ей можно передать `Entry`, так как `created` есть у всех «подтипов».

ПОДРОБНЕЕ

Для более точного понимания принципа подстановки стоит погрузиться в понятие вариантности, ☹️☹️☹️ но это большая отдельная тема. Мы пропустим её, но я оставлю несколько ссылок на тему в списке литературы.

Чтобы не нарушить принцип подстановки, в компонент `Record` надо передать «наименьший общий тип»:

```

type FinanceEntry = Income | Spend;
type RecordProps = { record: FinanceEntry };

const Record = ({ record }: RecordProps) => {
    // Дополнительные проверки не нужны,
    // в типе _точно_ содержится всё,
    // с чем функция может взаимодействовать.

    const sign = isIncome(record) ? "+" : "-";
    return `${sign} ${record.amount}`;
};

```

Принцип подстановки помогает определять, что и как можно компоновать. Его можно использовать как «интеграционный линтер», который подсвечивает преждевременные обобщения и неправильные абстракции.

Он также помогает находить места для *правильных* обобщений. Например, когда мы разделили задачу подсчёта денег на фильтрацию и суммирование, мы выделили функцию, которая может фильтровать не только по типу, но вообще как угодно:

```

type EntryPredicate = (record: Entry) => boolean;
type HistorySegment = List<Entry>;

type HistoryFilter = (
    history: RecordHistory,
    criterion: EntryPredicate
) => HistorySegment;

// Можно использовать не только эти функции:
const isIncome = ({ type }) => type === "Income";
const isSpend = ({ type }) => type === "Spend";

// Но и эти:
const beforeToday = ({ created }) => created < today;
const madeToday = ({ created }) => created >= today && created < tomorrow;

// А также эти:
const addedToday = (record) => isIncome(record) && madeToday(record);
const spentBeforeToday = (record) => isSpend(record) && beforeToday(record);

```

В `HistoryFilter` мы можем передать любую реализацию типа `EntryPredicate`. Такая гибкость в компоновке функциональности и есть главная польза принципа подстановки.

УПРОЩЕНИЕ

Мы не будем вдаваться в подробности того, как принцип подстановки связан с пред- и постусловиями и как они должны себя вести. Но я оставил несколько ссылок, которые рекомендую прочесть. ☺ ☺

-
1. "Functional Design Patterns" by Scott Wlaschin, <https://youtu.be/srOt1NAHYC0>
 2. Types and Typeclasses, Learn You Haskell, <http://learnyouahaskell.com/types-and-typeclasses>
 3. Generics in TypeScript, TypeScript Docs <https://www.typescriptlang.org/docs/handbook/2/generics.html>
 4. "A behavioral notion of subtyping" by Barbara H. Liskov, Jeannette M. Wing,
<https://dl.acm.org/doi/10.1145/197320.197383>
 5. The Principles of OOD, Robert C. Martin, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
 6. The Liskov Substitution Principle,
<https://web.archive.org/web/20151128004108/http://www.objectmentor.com/resources/articles/lsp.pdf>
 7. Covariance and Contravariance, Wikipedia,
[https://en.wikipedia.org/wiki/Covariance_and_contravariance_\(computer_science\)](https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science))
 8. "Why variance matters" by Ted Kaminski <https://www.tedinski.com/2018/06/26/variance.html>
 9. "The Ins and Outs of Generic Variance in Kotlin" by Dave Leeds, <https://typealias.com/guides/ins-and-outs-of-generic-variance/>
 10. "Design By Contract", c2.com, <https://wiki.c2.com/?DesignByContract>
 11. «Контрактное программирование» Тимур Шемсединов, https://youtu.be/K5_kSUvbGEO.

Архитектура

Успешному рефакторингу большого куска приложения может мешать слабая архитектура. Высокое зацепление между частями кода и беспорядочные зависимости мешают находить швы и изолировать изменения во время работы с кодом. А неочевидное взаимодействие частей приложения делает код запутанным и непонятным.

В этой главе мы поговорим о том, на что обращать внимание при рефакторинге большого куска приложения и как упростить задачу. Обсудим понятие архитектуры, её целей и пользы. Поговорим о доменной модели приложения, управлении зависимостями и взаимодействием кода с внешним миром. Затронем разницу между бизнес-логикой и UI-логикой.

ОСТОРОЖНО

Так как большая часть моего опыта связана с разработкой пользовательских приложений с богатой доменной логикой, я буду говорить об архитектуре такого «среднестатистического» приложения.

Разработка других видов программ может отличаться, а следовательно могут и отличаться подходы к построению архитектуры. Даже в пользовательских приложениях может найтись достаточно крайних случаев, которые могут вынудить писать код иначе.

Не про «папочки»

Архитектура — это взаимодействие частей системы; верхний уровень детализации приложения; выявление и приоритизация требований. ☹ ☹

В проектировании приложений не существует универсального решения, потому что особенности и требования у каждого проекта уникальны. Мы вряд ли сможем найти предметные области, которые бы не отличались друг от друга. То же и с проектами — два похожих приложения могут быть устроены по-разному из-за своих ограничений и требований.

Под требованиями мы будем иметь в виду функциональные, системные, бизнес- и другие требования. ☀ ☀ ☀

Мы не будем подробно останавливаться на разборе видов требований, их сборе, анализе и приоритизации — это отдельная большая тема. Вместо этого я оставлю ссылки на книги и статьи, в которых описано, как и зачем это нужно делать. ☀ ☀ ☀ ☀ ☀

Инструменты, о которых пойдёт речь дальше, — это не набор обязательных практик. Это скорее палитра, из которой можно выбирать подходящий инструмент под конкретную задачу. Ничего «обязательного» в архитектуре быть не может, каждый проект — это набор ограничений и компромиссов, баланс которых зависит от задачи.

Мы зашли так сильно издалека, чтобы подвести к одной простой мысли:

Проектирование ≠ раскладывание кода по папочкам

Папочки — это способ удобно исследовать код и открывать файлы в редакторе. Гораздо важнее, какие из выявленных требований критичны и как сделать так, чтобы организация кода не мешала развитию программы.

Требования дают понимание, как система должна себя вести в разных условиях. Это понимание уже подскажет детали организации: как должны друг с другом взаимодействовать части программы, что от чего должно зависеть, и какой уровень зацепления кода приемлем.

Цель приемлемой архитектуры в том, чтобы:

- не мешать разработке и не блокировать изменения программы;
- уменьшать количество решений, которые надо принимать прямо сейчас;
- стандартизировать написание кода и добавление фич;
- быть гибкой и оставлять пространство для изменений в будущем.

Не каждому приложению нужна пуленепробиваемая архитектура. Тратить время на улучшение архитектуры прототипа, который будет заброшен, бессмысленно. Но если приложение должно жить долго, а организация кода заметно мешает нам работать, можно попробовать выделить время на архитектурные улучшения.

ОДНАКО

Нет гарантий, что попытка поможет улучшить ситуацию. Иногда действительно проще переписать приложение с нуля, учтя предыдущий опыт и требования, которые стали известны за время жизни проекта. Но о решении «рефакторить или переписать» мы поговорим отдельно в одной из следующих глав.

В этой главе я не буду рассказывать, «как строить архитектуру с нуля» или «как быть в совсем запущенных случаях». Вместо этого я хочу показать небольшой набор приёмов, которые могут относительно недорого помочь с починкой наиболее общих проблем, которые могут встретиться в «среднестатистическом» приложении.

Моделирование бизнес-процессов

Главная ценность приложения в его бизнес-логике — тех процессах, которые приносят прибыль. Описание таких процессов в коде — это модель предметной области или *доменная модель*.

При рефакторинге приложения нам стоит обратить внимание, насколько чётко и однозначно определена эта модель. Могут ли разные разработчики определить из куска кода, о чём в нём идёт речь? Насколько точно они могут это сделать? Сколько будет разнотений в их мнениях?

Чем чётче составлена доменная модель, тем однозначнее разработчики будут её воспринимать. Это поможет им эффективнее общаться друг с другом и бизнесом при обсуждении задач и требований к проекту.

Повсеместный язык

В главе об именах переменных мы упоминали *повсеместный язык* (*ubiquitous language*)^{⊗ ⊕} — набор терминов, которыми пользуются «люди из бизнеса», когда описывают бизнес-процессы. При рефакторинге нам стоит следить, чтобы термины в коде соответствовали этому языку. Повсеместный язык уменьшает разногласия при чтении, потому что создаёт однозначные связи между термином и сущностью или процессом.

В примере ниже код нарушает этот принцип и использует разные термины для пользователя интернет-магазина:

```
const isMerchant = (user) => user.role === "seller";
const SellerPanel = ({ user }) => isMerchant(user) && <main>{/*...*/}</main>;
```

Из контекста непонятно, чем отличаются термины “seller” и “merchant”. Есть ли разница между сущностями, которые они описывают? Если да, то в чём она заключается?

Если термины значат одно и то же, лучше выбрать и оставить только один из них. Предпочтительнее выбрать тот, которым пользуется продукт-оунер или стейкхолдеры.

```
// Продукт-оунеры используют термин "seller",
// поэтому и мы тоже будем использовать его:

const isSeller = (user) => user.role === "seller";
const SellerPanel = ({ user }) => isSeller(user) && <main>{/*...*/}</main>;
```

Если термины означают разные вещи, то это повод проверить, правильно ли код отражает предметную область. Например, если “merchant” как-то связан с *компанией* пользователя, это могло бы вызвать вопрос, действительно ли в компоненте `SellerPanel` нужно проверять *пользовательскую* роль.

Доменная модель

Как мы упомянули ранее, доменная модель — это преобразования данных, отражающие бизнес-процессы. Доменные преобразования удобно выделять в отдельные функции, которые потом компоновать в цепочки преобразований данных.

[ПОДРОБНЕЕ](#)

Подробнее о цепочках преобразований мы говорили в главе о функциональном пайплайне.

Чтобы понять чем это полезно, посмотрим на пример. Допустим, перед нами функция онлайн-площадки для проведения аукционов. Функция создаёт объект нового аукциона и пытается добавить в него всё, что может произойти с аукционом за время его жизни:

[JAVASCRIPT](#)

```
function composeAuction(user, from, to, products, startPrice, invited) {
  return {
    created: true,
    author: user,
    timeRange: { from, to },
    lots: products.map(lotFromProduct),
    price: startPrice,
    participants: invited.filter(accepted),
    winners: [],
    bids: [],
    bestBid: null,
    open: false,
    expired: false,
    cancelled: false,
    closedLots: null
  };
}
```

Проблема функции `composeAuction` в том, что она не разделяет разные этапы жизненного цикла аукционов. Свежесозданный объект будет в *неопределённом* состоянии. Его структура не будет однозначно отражать этап жизненного цикла, в котором находятся данные.

В созданном объекте есть лишние, взаимоисключающие или противоречавшие поля. Например, пока аукцион не стартовал, вероятно, нет смысла добавлять историю ставок или текущий лот. Другой пример, флаги `open` и `expired` исключают друг друга, но из-за одновременного нахождения в объекте создаётся впечатление, что нет.

Структура созданного объекта недружелюбна к читателю. Без дополнительного контекста мы не сможем понять правил, по которым работают и меняются поля объекта. Нам нужна документация, чтобы понять, как живёт и эволюционирует такой объект аукциона.

При рефакторинге мы можем сперва выделить все возможные состояния, через которые проходит аукцион, а потом описать преобразования, которые будут приводить его в эти состояния. Например:

```
// Сперва мы создаём новый аукцион.
function createAuction(user, from, to, lots, startPrice) {
  return {
    author: user,
    timeRange: { from, to },
    lots: products.map(lotFromProduct),
    price: startPrice,

    // Поле `status` поможет валидировать объект и проверять,
    // можно ли с аукционом производить конкретные операции:
    status: STATUS.created,

    // Оно одно, поэтому разные состояния жизненного цикла
    // будут _однозначно_ отмечены в этом поле.
    // Мы не сможем создать аукцион, у которого одновременно
    // будет `STATUS.created` и `STATUS.expired`.
  };
}

// Допустим, приглашение участников – это отдельный процесс,
// тогда для него появится отдельная функция:
function inviteParticipants(auction, participants) {
  return {
    ...auction,
    participants,
    status: STATUS.inviting,
  };
}

// Проверим, может ли аукцион стартовать:
function canStart(auction) {
  const moment = Date.now();
  const { from, to } = auction.timeRange;
  return moment >= from && moment <= to;
}

// Когда аукцион запускается, остаются только участники,
// принявшие приглашение:
function startAuction(auction) {
  return {
    ...auction,
    participants: auction.participants.filter(accepted),
```

```

        currentLot: auction.lots.find(available),
        status: STATUS.active,
    };
}

// При каждой новой ставке, добавляем её в историю,
// (реализация может разниться в зависимости от concurrency):
function addBid(auction, bid) {
    return { ...auction, bids: [...auction.bids, bid] };
}

// При завершении аукциона:
function expireAuction(auction) {
    const { currentLot, ...rest } = auction;

    return {
        ...rest,
        closedLots: auction.lots.filter(isClosed),
        status: STATUS.expired,
    };
}

// При выводе участников, которые победили по лотам:
function defineWinners(auction) {
    return { ...auction, winners: participants.filter(ownsLot) };
}

// ...И так далее для всех доменных преобразований.

```

Функции доменной модели должны уметь «прокручивать» бизнес-процессы от начала и до конца, преобразовывая данные. Если мы можем это сделать, значит мы достаточно адекватно и полно отобразили предметную область в коде.

Да, кода стало больше. Но вместе с этим стало больше *информации* о том, как работает предметная область и какие данные в процессах участвуют. Стало и больше *ограничений*, которые программа должна учитывать. Например, мы явно говорим, что в незаконченном аукционе победителей быть не может, поэтому и поля `winners` там нет.

Чем больше информации о предметной области мы выразим в коде доменной модели, тем больше ошибок в дизайне мы найдём на этапе проектирования. Чем больше противоречий мы разрешим при проектировании, тем меньше будет нужды в дополнительных источниках правды и меньше противоречий между ними.

В языках со статической типизацией при рефакторинге мы также можем описать состояния данных в виде типов:

```

type CreatedAuction = {
    author: User;
    timeRange: LimitedTimeFrame;
    lots: ReadonlyList<Lot>;
    price: MonetaryValue;
    status: "created";
};

type PendingAuction = {};// ...
type ActiveAuction = {};// ...
type ExpiredAuction = {};// ...
type FinishedAuction = {};// ...

// При использовании Result для обработки ошибок
// мы можем также описать все потенциальные проблемы:

type ExpiredInvitation = "...";
type CantAddParticipantsAfterStart = "...";
type CantModifyExpiredAuction = "...";
// ...

type DomainError =
    | ExpiredInvitation
    | CantAddParticipantsAfterStart
    | CantModifyExpiredAuction;
// ...

```

Так мы быстрее заметим объекты и процессы, которые содержат в себе лишние данные или которым, наоборот, данных не хватает.

Направление зависимостей

Доменная модель — это самая важная часть приложения, потому что она описывает ключевые особенности проекта и несёт бизнес-ценность. Остальной код должен прислуживать ей и связывать её с внешним миром.

Код, описывающий пользовательские сценарии, лучше выражать в виде *Impureim-сэндвича*.[♂] В таком коде мы сперва получаем данные из «нечистых» источников, затем прогоняем их через цепочки преобразований, а потом сохраним или выведем на экран.

Об Impureim-сэндвиче и функциональном ядре в императивной оболочке мы детально говорили в главе о сайд-эффектах.

Связь с внешним миром

При рефакторинге большого куска приложения нам также стоит обращать внимание на зацепление кода со внешним миром. Зацепление может быть явным, когда мы используем сервис напрямую, привязываясь к его структуре. Но также зацепление может быть неявным.

Например, в интернет-магазине для создания заказа сервер просит в запросе от клиента объект с полем `products`. Так совпало, что в клиентском коде объект заказа тоже содержит поле `products`. Появляется желание использовать этот объект напрямую при отправке заказа:

JAVASCRIPT

```
function createOrder(user, products) {
  return { user, products };
}

// ...

async function sendOrder(order) {
  const response = await fetch("/api/orders/", {
    method: "POST",
    body: JSON.stringify(order),
  });
}

// ...
const order = createOrder(currentUser, productList);
await sendOrder(order);
```

Выглядит удобно, но мы только что создали точку зацепления с сервером. Мы теперь не можем изменить структуру объекта заказа внутри `createOrder`, потому что это поломает совместимость с сервером.

Это может быть пустяком для небольших проектов, которые редко меняют структуры данных или API. Но если нам важно масштабирование или гибкость API, или мы часто меняем названия полей в структурах данных на клиенте — это станет проблемой.

В таких ситуациях мы можем добавить анти-коррозионный слой (*Anti-Corruption Layer, ACL*) — набор преобразований, которые возьмут на себя ответственность за совместимость между API и нашим кодом.

ОСТОРОЖНО

Anti-Corruption Layer не следует путать с Access Control List. \circlearrowleft Оба понятия сокращаются как ACL, но отличаются по смыслу.

Анти-коррозионный слой может быть просто одинокой функцией адаптером, как `toServerOrder` в примере ниже:

JAVASCRIPT

```
function toServerOrder(clientOrder) {  
    // const serverOrder = ...  
    // Тут преобразования, необходимые для создания  
    // совместимой структуры данных для сервера.  
    return serverOrder;  
}  
  
async function sendOrder(order) {  
    const dto = toServerOrder(order);  
    // ...Отправляем на сервер уже подготовленный объект.  
}
```

...А может быть и отдельным модулем. Например, если фронтенду приходится работать с несколькими несовместимыми версиями API одновременно, то такой модуль может содержать логику подбора нужных структур под конкретную версию API.

JAVASCRIPT

```
function toServerOrder(clientOrder, config) {  
    const dataAdapter = orderAdapters[config.currentApiVersion];  
    const serverOrder = dataAdapter(clientOrder);  
    return serverOrder;  
}
```

Это освобождает фронтенд от необходимости постоянно проверять данные на соответствие серверной структуре *внутри приложения*. Нам всегда известно место, где эти данные подготовятся перед отправкой — клиент становится *отцепленным* от сервера.

При использовании статической типизации все преобразования и проекции данных можно выразить явно типами, чтобы разница между структурами была заметнее:

```

type Order = {
    user: User;
    products: List<Product>;
};

type ServerOrderDto = {
    userId: EntityId<User>;
    orderItems: List<OrderLine>;
};

type ServerOrderSelector = (clientOrder: Order) => ServerOrderDto;

```

Анти-коррозионный слой может увеличить сложность кода. Обычно его стоит добавлять места, где есть подозрение, что могут измениться данные или API: аналитика, логирование, сохранение данных на девайс пользователя и т.д.

К СЛОВУ

Чтобы понять, где может понадобиться анти-коррозионный слой, удобно использовать историю репозитория. Из репозитория мы можем собрать информацию о том, что и как часто в проекте уже менялось, что может измениться в будущем. Это помогает избежать больших переписываний кода и проблем с совместимостью в будущем.

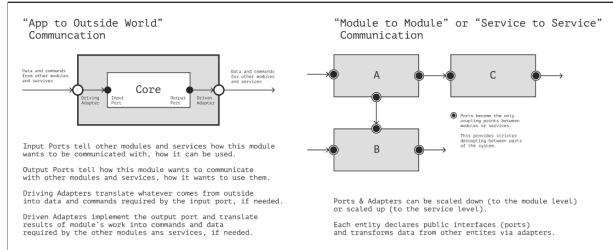
О технике «добычи метаданных» о проекте из репозитория подробно писал Адам Торнхил в книге «Your Code as a Crime Scene». ☀

Порты и адаптеры

Раз мы затронули тему адаптеров, стоит упомянуть и архитектурный паттерн «Порты и адаптеры». ☀ ☀

Под *адаптером* мы понимаем сущность, которая преобразует несовместимый интерфейс одного модуля к нуждам другого. *Port* — это спецификация того, как модуль хочет, чтобы с ним общались другие.

При использовании архитектурного стиля «Порты и адаптеры» взаимодействие модулей сводится к соединению портов одних модулей с адаптерами других. Так модули не знают о внутреннем устройстве друг друга, что уменьшает зацепление между ними.



При использовании их общение модулей сводится к соединению адаптеров с портами других модулей

Использование стиля «Порты и адаптеры» автоматически подталкивает к написанию кода в стиле Impureim-сендвича, потому что толкает «общение с чужаками» к краям приложения.

Использование этого подхода помогает уменьшить количество моков для тестов. Если мы везде используем один адаптер для запросов к сети, то проверить надо только его, а не мокать каждый запрос к сети. Это снижает необходимость обновлять тесты на каждое изменение кода, делая урон от тестов ниже.

Также порты и адаптеры купируют распространение изменений в пределах модуля. Место сочленения адаптера и интерфейса, который он адаптирует, служит барьером, на котором изменение должно остановиться.

ОДНАКО

Не любому проекту это может быть нужно. В простых приложениях, которые не собираются масштабироваться, порты и адаптеры могут добавить лишней работы без явной выгоды.

100500 видов архитектур

Мы не будем подробно рассматривать все придуманные человечеством способы «раскладывать код по папочкам». Я оставлю несколько ссылок на статьи и книги, которые считаю самыми полезными.

Для фронтенда отдельно хочу отметить Feature-Sliced. Ребята делают большую работу, пытаясь собрать в одном месте разный опыт разработки и выработать методологию дизайна фронтенд-приложений.

UI-логика

Отдельно во всём этом стоит UI. Разработка пользовательских интерфейсов — занятие хаотичное. UI меняется в среднем быстрее и чаще предметной области, но при этом пользователю в интерфейсе нужно отображать согласованные и корректные данные.

При работе с пользовательскими интерфейсами мы можем использовать такое правило:

Всегда держать UI-логику и бизнес-логику отдельно

Даже если данные в интерфейсе 1 в 1 совпадают с доменной моделью, лучше считать их *разными* данными. UI меняется быстрее, чем доменная модель — рано или поздно наборы этих данных начнут отличаться. Если мы не делаем разницы между бизнес и UI-логикой, этот момент можно упустить, и в доменных объектах окажутся данные из UI.

Например, представим, что нам нужно отрендерить корзину в UI, которая может быть свёрнута или развернута. В доменную модель может случайно просочиться кусок данных из UI:

JAVASCRIPT

```
const cart = {  
    products: [chocolateBar, sodaCan, teslaCar],  
    user: someUser,  
  
    // Это поле отвечает только за UI:  
    isExpanded: true,  
};
```

Проблема такого «просачивания» в том, что в объекте `cart` теперь смешиваются уровни абстракции и разные задачи. Это значит, что тестировать приложение с такими данными и готовить данные для отправки на сервер будет сложнее. Изменения в UI при наличии таких данных могут вызывать изменения всего приложения, часто неоправданные.

Более того, если не следить за разделением данных вовсе, то UI может метастазировать и в другие объекты:

```
const cart = {  
    // Детализируем пользователя, а там...  
    user: {  
        name: "Pippi Longstocking",  
        role: "buyer",  
  
        // ...Другие поля, которые нужны только в UI:  
        theme: SITE_THEME.light,  
    },  
  
    products: [chocolateBar, sodaCan, teslaCar],  
    isExpanded: true,  
};
```

Для решения этих проблем стоит отделять UI-логику от доменной:

```

const cart = {
  user: someUser,
  products: [chocolateBar, sodaCan, teslaCar],
};

const cartView = {
  isExpanded: true,
};

// Или, например:

const cartUi = {
  model: cart,
  view: cartView,
};

// Или, так:

const cartUi = {
  cart,
  ...cartView,
};

// Или даже так, если `cart` и `cartView`
// создаются отдельно друг от друга:

const cartUi = {
  ...cart,
  ...cartView,
};

```

Как именно компоновать объекты UI и домена, зависит от задачи. Главная идея в том, что данные бизнес-логики *в принципе* полезно отделять от данных презентационного слоя. Это расцепляет UI от домена и позволяет развивать их независимо друг от друга.

Реактивность

Когда речь заходит о UI-логике, стоит упомянуть реактивность. *Реактивные модели* мгновенно отображают изменения из UI в модели и обратно.  С реактивными интерфейсами соблазн объединить данные домена с UI-данными ещё выше, потому что реактивность создаёт иллюзию, будто бы эти данные — одно и то же.

При работе с реактивными интерфейсами удобно использовать паттерн MVVM. В нём view-model работает как медиатор между UI и бизнес-логикой, разделяя их и не давая смешаться.

В добавок к этому полезно использовать инструменты для *реактивных селекторов* или *мапинга данных*. Ими можно пользоваться для компоновки «обобщённого» презентационного слоя из доменных и UI данных:

TYPESCRIPT

```
// Очень хорошо с этим справляется Effectot.  
// Мы можем отдельно объявлять типы доменной логики:  
  
export type ConverterState = {  
    unit: Unit;  
    value: TimeStamp;  
};  
  
// ...И отдельно объявлять типы желаемых в UI данных:  
export type UiRepresentation = {  
    date: StringRepresentation;  
    stamp: NumberRepresentation;  
    open: boolean;  
};  
  
// Держать эти состояния мы тоже можем отдельно,  
// логика, соответственно, будет разделена и независима:  
const $converter = createStore<ConverterState>({ unit, value });  
const $isOpen = createStore<boolean>(false);  
  
// А для работы в UI уже можем использовать отображённый объект:  
const $representation = combine<UiRepresentation>(  
    $converter,  
    $isOpen,  
    (converter, isOpen) => ({  
        stamp: toMilliseconds(converter.unit, converter.value),  
        date: toDateString(converter.value),  
        open: isOpen,  
    })  
);
```

В итоге бизнес-логика становится изолирована, а в UI данные попадают через привязки в виде селекторов данных.

Тестирование

Результат рефакторинга архитектуры мы можем проверить по тому, насколько удобно тестировать код. Эвристику для этого мы позаимствуем из доклада "The Grand Unified Theory of Clean Architecture and Test Pyramid",  и заключается она в следующем:

- если мы можем протестировать бизнес-логику юнит-тестами;
- можем протестировать адаптеры интеграционными тестами;
- а юзкейсы или слайсы приложения — можем протестировать E2E тестами...

...То архитектура работает: код достаточно расцеплен, UI не влезает в бизнес-логику, времени и ресурсов на тестирование и поддержание тестов будет уходить меньше. 

1. "Software Architecture in Practice" by L. Bass, P. Clements, R. Kazman, https://www.goodreads.com/book/show/70143.Software_Architecture_in_Practice
2. "Enterprise Integration Patterns" by Gregor Hohpe, https://www.goodreads.com/book/show/85012.Enterprise_Integration_Patterns
3. "Designing Data-Intensive Applications" by Martin Kleppmann <https://dataintensive.net>
4. "Clean Architecture" by Robert C. Martin, <https://www.goodreads.com/book/show/18043011-clean-architecture>
5. "What I've Learned From Failure" by Reg Braithwaite, <https://leanpub.com/shippingsoftware/read>
6. "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
7. Software Requirements, Wikipedia, https://en.wikipedia.org/wiki/Software_requirements
8. List of System Quality Attributes, Wikipedia, https://en.wikipedia.org/wiki/List_of_system_quality_attributes
9. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
10. "Ubiquitous Language" by Martin Fowler, <https://martinfowler.com/bliki/UbiquitousLanguage.html>
11. "Impureim Sandwich" by Mark Seemann, <https://blog.ploeh.dk/2020/03/02/impureim-sandwich/>
12. Anti-corruption Layer Pattern, Microsoft Docs, <https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer>
13. Access Control List, Википедия, <https://ru.wikipedia.org/wiki/ACL>
14. "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>
15. "Functional Architecture is Ports and Adapters" by Mark Seemann, <https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/>

16. "Ports & Adapters Architecture" by Herberto Graça, <https://herbertograca.com/2017/09/14/ports-adapters-architecture/>
17. "Test-Induced Design Damage" by David Heinemeier Hansson, <https://dhh.dk/2014/test-induced-design-damage.html>
18. "The Software Architecture Chronicles" by Herberto Graça, <https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/>
19. "Functional architecture: The pits of success" by Mark Seemann, <https://youtu.be/US8QG9I1XW0>
20. Feature-Sliced Design, Architectural methodology for frontend projects, <https://feature-sliced.design>
21. Реактивное программирование, Википедия, https://ru.wikipedia.org/wiki/Реактивное_программирование
22. Model-View-ViewModel, Википедия, <https://ru.wikipedia.org/wiki/Model-View-ViewModel>
23. View Models, Reactive UI, <https://www.reactiveui.net/docs/handbook/view-models/>
24. "The Grand Unified Theory of Clean Architecture and Test Pyramid" by Guilherme Ferreira, <https://youtu.be/gHSpj2zM9Nw>
25. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>

Декларативность

В главе об абстракции мы обсудили декомпозицию задач, а также разделение в коде намерения и реализации. Мы рассмотрели, как декомпозиция помогает улучшать читаемость кода и акцентировать внимание на деталях, которые важны в конкретный момент времени.

В этой главе мы расширим идеи абстракции и поговорим о декларативности. Мы обсудим, что такое декларативный код, как его писать и в чём его польза и преимущество по сравнению с кодом в императивном стиле.

Читаемость

Чтобы лучше понять пользу декларативности, сперва обсудим разницу между декларативным и императивным кодом. Посмотрим на два фрагмента ниже:

```
// 1.

function keepEvenNumbers(array) {
  const result = [];

  for (const x of array) {
    if (x % 2 === 0) {
      result.push(x);
    }
  }

  return result;
}

// 2.

function keepEvenNumbers(array) {
  return array.filter((x) => x % 2 === 0);
}
```

Обе функции фильтруют переданный массив чисел, оставляя только чётные. Разница между ними в том, как они это делают. Первая функция описывает, *как* решить задачу:

- Создать пустой массив `result`;
- Проитерировать переданный массив `array`;
- Для каждого элемента проверить, чётный ли он;
- Если да, добавить его в `result`.

Вторая функция описывает, *что* надо сделать. Она акцентирует внимание на *критериях* фильтрации, а не деталях её алгоритма, которые скрыты внутри метода `filter`.

В этом и есть разница между императивным и декларативным стилем кода. Декларативный код описывает, *что* нужно сделать, императивный — *как* это сделать.

Код в императивном стиле зачастую читать сложнее, потому что он смешивает намерение и детали реализации. Декларативный стиль, наоборот, подталкивает к декомпозиции задач и разделению кода по уровням абстракции. Названия функций и переменных в декларативном коде несут больше смысловой нагрузки, поэтому код становится проще читать.

Для примера посмотрим на функцию `validate` во фрагменте кода ниже. Тело функции пестрит деталями, а её название мало говорит о цели функции. Такой код сложно понять сходу:

```

function validate(user, cart) {
  return (
    !!cart.items.length &&
    user.account >= cart.items.reduce((tally, item) => tally + item.price, 0)
  );
}

```

Чтобы решить эту проблему, мы можем декомпозировать задачу и разделить её части по уровням абстракции. Например, детали различных проверок выделим в отдельные функции, названия которых будут отражать суть операций:

```

// cart.js
function isEmpty(cart) {
  return !cart.products.length;
}

function totalPriceOf(cart) {
  return cart.items.reduce((tally, item) => tally + item.price, 0);
}

// user.js
function canAffordSpending(user, amount) {
  return user.account >= amount;
}

```

Названия выделенных функций теперь используют термины, более подходящие для описания их целей. Это помогает управлять вниманием читателя при использовании этих функций внутри

`validate`

```

// order.js

function validate(user, cart) {
  return !isEmpty(cart) && canAffordSpending(user, totalPriceOf(cart));
}

```

Об уровнях абстракции, переключении между ними и управлении вниманием читателя мы говорили более детально ранее в главе об абстракции.

Также названия выделенных функций теперь несут в себе часть информации о смысле функции

`validate`. Поэтому мы можем заменить имя `validate` на более информативное, например, `canMakeOrder`. Тогда код функции превратится в «текст», похожий на обычное предложение:

JAVASCRIPT

```
// order.js

function canMakeOrder(user, cart) {
    const orderPrice = totalPriceOf(cart);
    return !isEmpty(cart) && canAffordSpending(user, orderPrice);
}

// The (user) (canMakeOrder) IF the (cart) (!isEmpty)
// AND they (canAffordSpending) (orderPrice) of money.
```

Декларативный код больше похож на то, как люди общаются друг с другом в жизни, поэтому его проще воспринимать. Он сообщает полезную информацию, но не перегружает читателя лишними деталями. Такое «общение» вежливое и ненарочитое, читатели будут меньше от него уставать.

ОДНАКО

В небольших проектах дотошная декомпозиция может оказаться не такой важной. Если кода мало, читатели от него устают меньше. Необходимость декомпозиции будет зависеть оттого, насколько команде просто читать и работать с имеющимся кодом.

Надёжность

Следующий раздел спорный и субъективный, но по моему опыту в императивном коде проще допустить случайные ошибки. Отчасти потому что в нём приходится одновременно думать о «цели» и «способе её достичь», а ещё потому что императивный код зачастую обёмнее и статистически в нём вероятнее допустить ошибку. ☹

Например, посмотрим на функцию выбора математической операции `selectOperation`:

```
function selectOperation(kind) {
  let operation = null;
  switch (kind) {
    case "log":
      operation = (x, base) => Math.log(x) / Math.log(base);
    case "root":
      operation = (x, root) => x ** -root;
    default:
      operation = (x) => x;
  }
  return operation;
}
```

В каждом блоке `case` этой функции пропущена инструкция `break`, поэтому переменная `operation` всегда будет равна `(x) => x`. Подобную ошибку относительно просто заметить в небольшой функции, но если кода много, её гораздо легче пропустить.

Мы можем улучшить код, использовав `return` внутри блоков `case`, тем самым перестав зависеть от внутренней переменной:

```
function selectOperation(kind) {
  switch (kind) {
    case "log":
      return (x, base) => Math.log(x) / Math.log(base);
    case "pow":
      return (x, power) => x ** power;
    default:
      return (x) => x;
  }
}
```

Но это не решит проблему со случайными ошибками, а только замаскирует её. Во фрагменте выше мы, например, можем случайно пропустить `return`, и функция будет работать неправильно. Избавиться от проблемы можно, сделав выбор декларативным:

```

const log = (x, base) => Math.log(x) / Math.log(base);
const pow = (x, power) => x ** power;
const id = (x) => x;

function selectOperation(kind) {
  const operations = { log, pow, id };
  return operations[kind] ?? operations.id;
}

```

В коде выше мы поручаем «выбор» операции *механизмом языка*. Мы указываем объект с данными и критерий выбора, а интерпретатор JavaScript сам находит нужное значение в объекте по указанному ключу. Нам не важно, как будет сделан этот выбор, важен лишь его результат — это и есть декларативность.

ВКУСОВЩИНА

Последний фрагмент мне нравится ещё и по эстетическим причинам. Выбор из объекта по ключу выглядит более натуральным решением этой задачи, в то время как `switch` кажется шумным и многословным.

Расширяемость

Расширять императивный код часто сложнее.

В императивном коде при добавлении новой фичи нам надо не только понять, что добавить, но также где и как это добавить. Расширение функциональности декларативного же кода зачастую ограничивается обновлением «настроек» алгоритма.

К СЛОВУ

Идею для следующего примера я подсмотрел в лекции Тимура Шемсединова о декларативном стиле и метапрограммировании. Рекомендую к просмотру. ☺

Для примера сравним две реализации функции `parseDuration`, которая преобразует форматированную строку с периодом времени в количество миллисекунд в нём. В первой версии алгоритм реализован императивно:

```
/** @example parseDuration('1h 25m 16s') === 5_116_000 */
function parseDuration(stringRepresentation) {
    const s = stringRepresentation;
    if (typeof s !== "string") return 0;

    const seconds = s.match(/(d+)s/);
    const minutes = s.match(/(d+)m/);
    const hours = s.match(/(d+)h/);

    let duration = 0;
    if (seconds) duration += +seconds[1] * 1000;
    if (minutes) duration += +minutes[1] * 1000 * 60;
    if (hours) duration += +hours[1] * 1000 * 60 * 60;
    return duration;
}
```

...Во второй версии — декларативно:

```

// Вся информация о поддерживаемом
// формате строки вынесена в объект:
const MULTIPLIER = {
  s: 1000,
  m: 1000 * 60,
  h: 1000 * 60 * 60,
};

// Шаги алгоритма представлены отдельными функциями:
const sumDurations = (sum, [value, unit]) => sum + value * MULTIPLIER[unit];
const hasValidValue = ([value]) => !Number.isNaN(value);

const parseComponent = (component) => {
  const value = +component.slice(0, -1);
  const unit = component.slice(-1);
  return [value, unit];
};

/** @example parseDuration('1h 25m 16s') === 5_116_000 */
function parseDuration(stringRepresentation) {
  if (typeof stringRepresentation !== "string") return 0;

  const components = stringRepresentation.split(" ");
  return components
    .map(parseComponent)
    .filter(hasValidValue)
    .reduce(sumDurations, 0);
}

```

Второй вариант проще расширить, потому что в нём *разделён редко и часто меняющийся код*. Если мы захотим расширить формат строки, например, добавив дни и недели, нам потребуется обновить только объект `MULTIPLIER`. Остальной код функции останется неизменным.

Так мы будто вынесли в объект `MULTIPLIER` «настройки» алгоритма, отделив их от «логики» задачи. Теперь поддерживаемый формат строки более явный, а сам алгоритм гибче, потому что может работать с любыми значениями из этого объекта. «Настройки» больше не «вшиты» в код алгоритма.

К СЛОВУ

Такое поведение — когда добавить однотипную функциональность можно не меняя существующий код — цель принципа открытости-закрытости из SOLID. ☺

Расширение формата строки теперь ограничится добавлением новых полей для объекта

`MULTIPLIER`. Код самого алгоритма меняться не будет:

JAVASCRIPT

```
const MULTIPLIER = {  
    s: 1000,  
    m: 1000 * 60,  
    h: 1000 * 60 * 60,  
  
    // Добавили дни и недели:  
    d: 1000 * 60 * 60 * 24,  
    w: 1000 * 60 * 60 * 24 * 7,  
};  
  
// Так как остальной код функции остался прежним,  
// вероятность допустить случайную ошибку или опечататься ниже.  
// Кроме этого по выделенным в `MULTIPLIER` «настройкам»  
// проще автоматически сгенерировать данные для тестов.
```

В первой реализации нам бы потребовалось менять код всей функции `parseDuration`:

```

function parseDuration(stringRepresentation) {
  const s = stringRepresentation;
  if (typeof s !== "string") return 0;

  const seconds = s.match(/(d+)s/);
  const minutes = s.match(/(d+)m/);
  const hours = s.match(/(d+)h/);
  const days = s.match(/(d+)d/);
  const weeks = s.match(/(w+)d/);

  let duration = 0;
  if (seconds) duration += +seconds[1] * 1000;
  if (minutes) duration += +minutes[1] * 1000 * 60;
  if (hours) duration += +hours[1] * 1000 * 60 * 60;
  if (days) duration += +days[1] * 1000 * 60 * 60 * 24;
  if (weeks) duration += +weeks[1] * 1000 * 60 * 60 * 24 * 7;
  return duration;
}

// К слову о том, насколько просто в императивном коде допустить случайную ошибку
// печеньку тем, кто заметил в регулярном выражении для `weeks` опечатку :-)

```

ОДНАКО

Подобное выделение «настроек» хорошо помогает для расширения кода *однотипной* функциональностью. Если требуется добавить новое поведение в сам алгоритм, это может не помочь.

Стоит помнить, что такое «метапрограммирование» нужно не всегда, потому что *обобщённые* функции могут быть сложнее.

Как правило, декларативное обобщение полезно, когда мы замечаем часть функции, которая «меняется слишком часто» по сравнению с остальным кодом. Такую функциональность можно сделать декларативной, это уменьшит вероятность случайных ошибок при её обновлении.

Конфигурируемость

В предыдущем примере мы вынесли «настройки» алгоритма в отдельный объект. Мы мотивировали это тем, что настройки и код меняются с разной скоростью, поэтому их лучше держать разделёнными.

На самом деле это одно из известных правил, которых рекомендуют придерживаться в методологии двенадцати факторов (12-Factor Apps). ☀ Это правило можно описать как:

Всегда держать конфиги отдельно от кода

Настройки и конфигурация обычно меняются чаще кода, который они настраивают. Если конфиги «вшиты» в код, их изменение будет опаснее и сложнее, чем когда они отделены.

Кроме этого захардкоженная конфигурация затрудняет смену окружения, в котором запускается программа. Например, если наше приложение должно запускаться в тестовом и продакшен окружении, а настройки деплоя или подключения к сторонним сервисам типа базы данных или API захардкожены, то для смены окружения понадобится менять значения этих настроек руками.

К СЛОВУ

Именно этот признак помогает находить конфиги среди остального кода: если значение переменной зависит от окружения или среды запуска — это точно часть конфигурации.

Посмотрим на антипример. Допустим, базовый URL API в коде ниже должен меняться для разных окружений. В функции `fetchUser` базовый URL «зашит» прямо в теле функции:

JAVASCRIPT

```
async function fetchUser(id) {  
  const response = await fetch(`https://api.our-app.com/v1/users/${id}`);  
  const data = await response.json();  
  return data.user;  
}  
  
// Любой вызов `fetchUser` обратится к конкретной версии API – api.our-app.com.  
// Чтобы поменять окружение, нужно изменить код функции.
```

Выделять конфигурацию удобнее всего, следуя *приоритету трансформаций* (*Transformation Priority Premise, TPP*). ☀ Сперва стоит выделить значения конфигов в локальные переменные, а потом — в переменные окружения или файлы конфигурации.

```
// Шаг 1: вынести конфиги в локальные переменные.
const baseUrl = "https://api.our-app.com";
const apiVersion = "v1";

async function fetchUser(id) {
  const response = await fetch(`.${baseUrl}/${apiVersion}/users/${id}`);
  const data = await response.json();
  return data.user;
}

// Шаг 2: утащить их в настоящий конфиг.
// Это могут быть как переменные окружения, так и отдельные модули.
// Главное, чтобы между кодом и конфигами было _чёткое_ разделение.
import { networkConfig } from "@config";

async function fetchUser(id) {
  const response = await fetch(`.${networkConfig.apiRoot}/users/${id}`);
  const data = await response.json();
  return data.user;
}
```

Автоматное программирование

Как мы упоминали в предыдущей главе, бизнес-логику приложения и UI-логику лучше держать отдельно. Код бизнес-логики должен отвечать за бизнес-процессы и преобразования данных, связанные с ними. Код UI-логики — за рендер пользовательского интерфейса.

Иногда UI-логика бывает сложной. Например, если она описывает поведение взаимозависимых компонентов или рендер динамических кусков интерфейса, зависящих от большого количества условий.

Чтобы код сложной UI-логики оставался читабельным, мы можем представлять UI в виде конечного набора его состояний. Каждое такое состояние описывает интерфейс, который видит пользователь, и условия, в которых мы рендерим его на экране.

Взаимодействие пользователя с UI тогда можно описать, как последовательность таких состояний. Если количество состояний ограничено, то такую последовательность мы можем назвать *конечным автоматом (Finite State Machine)*,  а способ её запрограммировать — *автоматным программированием*.

Конечный автомат — это математическая концепция, но она хорошо ложится на описание UI как функции от состояния данных. ☺ Она побуждает разделять данные (состояние) и эффекты (рендер UI) и помогает делать работу с интерфейсом более детерминированной.

Основная идея конечного автомата в ограниченности набора состояний и в однозначных правилах переходов между ними. Например, UI интернет-магазина во время оформления заказа можно выразить в виде такого набора состояний и потенциальных переходов:

«Юзкейс оформления заказа»:

Текущее состояние	Возможные следующие состояния	
OrderPage	MainPage	Confirming
Confirming	Success	Failure
Success	MainPage	
Failure	MainPage	OrderPage

Мы можем представить такой автомат в виде диаграммы переходов между состояниями:

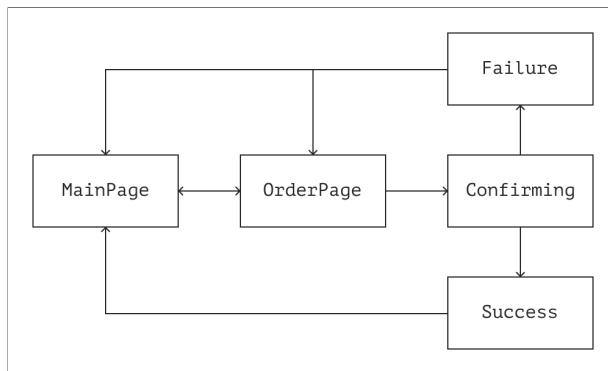


Диаграмма переходов между состояниями в интернет-магазине

Но также мы можем представить этот автомат в коде как коллекцию состояний и переходов:

```
const fsm = createMachine({
  states: {
    main: {}, // Переходы из ` MainPage`...
    order: {}, // Из ` OrderPage`...
    confirming: {}, // Из ` Confirming`...
    success: {}, // Из ` Success`...
    failure: {}, // Из ` Failure`...
  },
});
```

Сами состояния можем описать в виде, например, компонентов:

```
// Отражает состояние `OrderPage`:
const ConfirmOrder = () => (
  <form onSubmit={fsm.to("confirming")}>{/*...*/}</form>
);

// Отражает состояние `Success`:
const OrderConfirmed = () => (
  <>
    Order Confirmed.
    <a href={fsm.to("main")}>Back to main page</a>
  </>
);
;

// Отражает состояние `Failure`:
const OrderError = () => (
  <>
    Couldn't confirm the order.
    <a href={fsm.to("main")}>Back to main page</a>
    <a href={fsm.to("order")}>Try again</a>
  </>
);
;

// Отражает состояние `Confirming`:
const Confirming = () => "Loading...";
```

Тогда входная точка приложения может использовать этот автомат, чтобы решать, что рендерить на экране:

```
// Выбирает компонент по текущему состоянию интерфейса:
function Checkout() {
  const [state] = useMachine(fsm);

  return state.match
    .with("confirming", Confirming)
    .with("success", OrderConfirmed)
    .with("failure", OrderError)
    .orElse(ConfirmOrder);
}
```

Польза конечных автоматов в том, что из одного состояния можно перейти *только в определённый* набор следующих. В состояния вне этого списка попасть нельзя. Это помогает сделать работу с UI более декларативной и детерминированной.

Кроме этого, используя конечный автомат, проще отделять UI-данные от данных бизнес-логики. Как правило, у состояний автомата есть идентификаторы, которые соотносят их с результатом на экране. По этим идентификаторам мы всегда можем определить данные, которые относятся только к UI, и не смешивать их с данными бизнес-логики.

ИНСТРУМЕНТЫ

Сам автомат и логика переходов между состояниями может быть описана с использованием различных библиотек. В примере выше я использовал вымышленный инструмент, чтобы не претендовать на «каноничность решения». Но как хороший пример библиотеки для работы с конечными автоматами в UI могу предложить xstate.[♂]

Налог на декларативность

Как мы упоминали выше, у декларативного стиля кода есть недостатки.

Сложность поддержки

Обобщения могут сделать код сложнее. За декларативным «фасадом» может скрываться чрезмерно сложная абстракция, которую тяжело понимать другим разработчикам.

После рефакторинга нам всегда следует проверять, действительно ли изменения кода пошли на пользу. Если код стало тяжелее читать или поддерживать, изменения лучше откатить.

Когда у нас есть сомнения в лёгкости поддержки, мы можем запросить реview на изменения от большего чем обычно количества разработчиков. Так мы узнаем, понятен ли код команде и легко ли его поддерживать без помощи авторов изменений.

Производительность

Императивный код, как правило, производительнее. В тех местах, где производительность важнее читаемости, декларативностью можно пожертвовать.

При этом может быть полезно изолировать «островки императивности» от остального кода.

Например, если для работы приложения нам нужен какой-то производительный алгоритм, его реализацию можно описать императивно внутри функции:

JAVASCRIPT

```
function mergeTrees(treeA, treeB) {  
    // ...Быстрая императивная реализация.  
}
```

...А остальную часть приложения описать декларативно:

JAVASCRIPT

```
function mergeCompanyDepartments(departmentIdA, departmentIdB) {  
    return mergeTrees(  
        extractDepartment(departmentIdA),  
        extractDepartment(departmentIdB)  
    );  
}
```

Так мы «спустим» императивную реализацию «на уровень ниже», изолируем её от остального кода, а имя функции сделаем декларативным описанием всего алгоритма целиком.

-
1. "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>
 2. «Метапрограммирование и мультипарадигменное программирование» Тимур Шемсединов, <https://youtu.be/Bo9y4IxdNRY>
 3. The Principles of OOD, Robert C. Martin, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
 4. 12 Factor Apps, <https://12factor.net>
 5. Transformation Priority Premise, Wikipedia https://en.wikipedia.org/wiki/Transformation_Priority_Premise
 6. Конечный Автомат, Википедия, https://ru.wikipedia.org/wiki/Конечный_автомат
 7. Управление состоянием приложения с помощью конечного автомата, <https://bespoyasov.ru/blog/fsm-to-the-rescue/>
 8. JavaScript and TypeScript finite state machines and statecharts, XState, <https://github.com/statelyai/xstate>

Статическая типизация

В прошлых главах, говоря о читабельности, мы в основном обсуждали переменные, функции и модули. Мы иногда уделяли внимание особенностям статически типизированных языков, но не погружались в подробности.

Однако, статическая типизация тоже может быть инструментом для написания более выразительного кода. Мы можем использовать типы и интерфейсы для передачи дополнительной информации читателю или для наглядного проектирования приложения.

В этой главе мы обсудим, как передавать больше знаний о предметной области через типы и «запрещать» невалидные преобразования данных. Рассмотрим, как использовать повсеместный язык в сигнатурах функций и выслеживать с помощью типов ошибки в дизайне API.

ПЕРЕД НАЧАЛОМ

В тексте мы не будем обсуждать, *нужна ли* статическая типизация. Вместо этого мы сосредоточимся на том, как использовать типы в качестве ещё одного инструмента для рефакторинга.

Статическая типизация вызывает много споров и нравится не всем, это нормально. Если вашему проекту типизация не подходит, эту главу можно пропустить.

Повсеместный язык

В “Domain Modeling Made Functional” Скотт Влашин предостерегает читателей от одержимости примитивами.^{♂ ♂} Чтобы избежать её, он предлагает использовать типы для описания предметной области.

Идея замены примитивов на доменные типы может вызывать споры в команде. Перед применением обсудите эту идею с другими разработчиками.

Типы данных и сигнатуры функций могут нести дополнительную информацию о задаче, которую код решает. Они могут отражать контекст задачи, взаимодействие сущностей и даже модель работы бизнес-процессов. При вдумчивом использовании типы могут стать альтернативой документации:

TYPESCRIPT

```
// Примитивные типы не передают контекста задачи,
// в них не достаёт деталей предметной области:

type Account = {
    date: string;
    user: number;
    value: number;
};

// Описание доменных типов помогает передать
// отношения между сущностями и принципы работы домена:

type Account = {
    date: DateTimeIso;
    user: UserId;
    value: MoneyAmount;
};
```

Когда названия типов используют термины из предметной области, они составляют часть *повсеместного языка (ubiquitous language)*^{⊗ ⊗}. Этим языком выражаются владельцы бизнеса и люди, которые непосредственно связаны с бизнес-процессами.

ПОДРОБНЕЕ

Более детально о повсеместном языке мы говорили в главах об именах сущностей и архитектуре.

Польза повсеместного языка в его однозначности. Если вся команда, включая «неразработчиков» выражается одинаковыми терминами, вероятность потерять «при переводе с языка бизнеса на программистский» ниже. Баги и неправильную работу доменной модели на ранних этапах заметить проще.

Моделирование домена

Во многих статически типизированных языках преобразования данных удобно выражать через функциональные типы. ☈ Совокупность таких типов может описывать процессы предметной области — моделировать домен. Польза такой модели в том, что цена ошибки в ней ниже, чем при реализации бизнес-процессов в коде.

Типы помогают составить «верхнеуровневое» понимание работы системы. В такой модели видно взаимодействие её частей, контракты модулей и используемые данные. Но также в ней видны ошибки проектирования и несоответствия модели реальному миру.

Ошибки в типах исправить проще, чем ошибки в реализации. Так мы можем проектировать работу приложения до того, как приступить к реализации. Пишем черновик, смотрим на недостатки модели, переделываем и повторяем проверку:

```

// Опишем данные, с которыми работает приложение.
// Укажем различные состояния, через которые они проходят
// на разных стадиях жизненного цикла приложения:

type CreatedOrder = {
    createdAt: TimeStamp;
    user: UserId;
    items: ProductList;
};

type ValidatedOrder = {
    /*...*/
};

type DiscountedOrder = {
    /*...*/
};

type Order = CreatedOrder | ValidatedOrder | DiscountedOrder;

// Спроектируем процессы предметной области,
// которые преобразуют эти данные:

type CreateOrder = (user: UserId, items: ProductList) => CreatedOrder;
type ValidateOrder = (order: CreatedOrder) => ValidatedOrder;
type ApplyDiscount = (order: ValidatedOrder, value: Price) => DiscountedOrder;

// Если мы заметим, что в каком-то из типов есть ошибка,
// (например, процесс отличается от того, что происходит в реальности)
// мы можем быстро и относительно дёшево исправить модель:

type ApplyDiscount = (
    order: ValidatedOrder,
    coupon: DiscountCoupon
) => DiscountedOrder;

```

Во время рефакторинга такие проверки помогают обнаружить бизнес-процессы, которые не отвечают требованиям проекта или не учитывают ограничений предметной области:

```

type Divider = (a: number, b: number) => number;
const divide: Divider = (a, b) => a / b;

// Принимает ли тип 'divide' вторым аргументом ноль?
// Стоит ли это разрешить?
// Если да, как обработать деление на ноль?
// Нужно ли возвращать контейнер из этой функции?

// Мы можем обогатить доменную модель
// дополнительными данными и ограничениями:

type RealNumber = // ...Допускает любое число.
type NaturalNumber = // ...Допускает число больше нуля.

type Divider = (a: NaturalNumber, b: NaturalNumber) => RealNumber;
const divide: Divider = (a, b) => a / b;

// Теперь из типа 'Divider' ясно, что деление на ноль
// уже должно быть обработано в вызывающем коде.

```

Типизация в TypeScript

В TypeScript для моделирования предметной области и создания доменных типов мы можем использовать несколько вариантов:   

- Тайп-алиасы;
- Классы;
- «Брендированные» типы.

К СЛОВУ

Из-за структурной типизации в TypeScript удобство и применимость каждого из перечисленных вариантов может отличаться. 

В книге мы не будем вдаваться в нюансы и тонкости системы типов этого языка. Вместо этого мы сосредоточимся на пользе типов в рефакторинге кода.

Однако для лучшего понимания ограничений структурной типизации я оставлю в тексте несколько ссылок на эту тему.   

Самый простой, но при этом самый ненадёжный способ создания доменных типов — использовать тайп-алиасы.  С их помощью удобно давать примитивным типам информативные имена, но сложно передавать ограничения предметной области. Например, такой код синтаксически вполне валиден, а с точки зрения домена — нет:

```
// Тайп-алиас может, например, дать примитиву полезное имя,
// которое будет отражать смысл типа согласно домену:
type RealNumber = number;
type NaturalNumber = number;

// Но из-за особенностей типизации
// мы можем получить невалидную доменную модель:
const x: RealNumber = -1;
const y: NaturalNumber = x;

// Упс!
// «-1» не может быть натуральным числом.
```

В тайп-алиасах сложно выразить ограничения домена и настроить «валидацию» присваиваемых значений, поэтому нет гарантий, что в `NaturalNumber` окажется именно натуральное число:

```
function divide(a: NaturalNumber, b: NaturalNumber): RealNumber {
    return a / b;
}

// Компилятор доволен, а в рантайме ошибка:
divide(1, 0);
```

Поэтому если нам нужна валидация или проверка на несопоставимость разных типов, придётся использовать классы или брендированные типы:  

```
// При использовании классов
// мы можем добавить валидацию значений в конструктор:

class NaturalNumber {
    constructor(value) {
        if (value <= 0 || Math.floor(value) !== value) {
            throw new Error("The value must be a positive integer.");
        }

        this.value = value;
    }
}

// Тогда создать «неправильное» значение не получится:

new NaturalNumber(-1); // Error!
new NaturalNumber(42); // NaturalNumber

// Но классы довольно многословные
// и ими не удобно пользоваться, например,
// если нужно сделать обёртку над примитивом.

// Создавать «числа» через new NaturalNumber(42),
// а потом как-то реализовать арифметические операции
// с такими значениями – накладно и требует много кода.

// Второй вариант – использовать брендирование типов:

type Tagged<T, S> = T & { __tag: S };
type NaturalNumber = Tagged<number, "natural">;

// А значения создавать только через специальные фабрики:

function naturalFrom(value: number): NaturalNumber {
    // Всю валидацию значений можно будет реализовать
    // внутри такой функции-фабрики:

    if (value <= 0 || Math.floor(value) !== value) {
        throw new Error("The value must be a positive integer.");
    }

    return value as NaturalNumber;
}
```

```
}

naturalFrom(-1); // Error!
naturalFrom(42); // NaturalNumber
```

Проблема классов и «брендирования» в том, что за правильностью их использования необходимо следить. То есть понадобится писать правила для линтера или искать ошибки во время код-ревью. Это ненадёжно.

Здесь сложно порекомендовать конкретный способ, всё зависит от проекта и нужд команды. Однако, можем отметить, что для чисто описательных целей даже тайп-алиасы вполне подходят. Часто доменной модели, построенной на тайп-алиасах, уже достаточно, чтобы найти необнаруженные ранее ошибки в проектировании.

Сверка модели с реальностью

Бизнес-процессы преобразуют данные из одного состояния в другое. Типы могут помочь синтаксически зафиксировать эти состояния и дать им названия. Когда каждый шаг преобразования как-то назван, нам проще рассуждать о процессе целиком и находить ошибки в его логике.

В примере ниже функция `sendRecoverLink` работает с объектом типа `User`. У этого типа есть флаг `verified`, но нет никаких правил, объяснявших бы, когда и почему этот флаг принимает значение `true`:

TYPESCRIPT

```
type User = {
  id: string;
  verified?: boolean;
};

async function sendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}
```

Из-за текущей реализации `User` функция `sendRecoverLink` принимает данные, которые в половине случаев для неё невалидны. Мы можем перестраховаться от невалидной передачи данных, затруднив её на уровне типов.

Верификация пользователя — вероятно, отдельный бизнес-процесс, в результате которого объект пользователя становится верифицированным. Этую причинно-следственную связь можно выразить прямо в типах, если разделить типы верифицированных и неверифицированных пользователей:

```
// Описываем типы верифицированных и неверифицированных пользователей,
// как состояния, через которые проходят данные:

type CreatedUser = { name: string };
type VerifiedUser = { name: string; verified: true };

type User = CreatedUser | VerifiedUser;

// Показываем причинно-следственную связь, как тип процесса верификации.
// Рассказываем, через какие шаги проходят данные, и в результате чего
// пользователь становится верифицированным:

type VerifyUser = (user: CreatedUser) => Promise<VerifiedUser>;

// Фиксируем ограничения на восстановление пароля –
// восстановить пароль может только верифицированный пользователь:

type RecoverPassword = (user: VerifiedUser) => Promise<void>;

// Теперь выслать ссылку на восстановление пароля
// неверифицированному пользователю будет нельзя.
// Невалидную операцию выразить в коде станет сложнее:

const sendRecoverLink: RecoverPassword = async (user) => {
  await api.recoverPassword(user.id);
};

sendRecoverLink(unverifiedUser); // Error!
```

Опять же, в TypeScript добиться бронебойной невыразимости невалидных процессов сложно, в других типизированных языках это может быть проще. Но даже просто фиксация разных состояний данных в типах помогает замечать ошибки в логике бизнес-процессов на этапе проектирования или реью.

В типах подобные ошибки замечать проще, потому что выражать условия в типах сложнее, чем в реализации. Это заставляет описывать процессы декларативно и прямолинейно. И если в логике преобразования заметна неоднозначность, это может стать сигналом проблем.

Нарушения договорённостей

Благодаря явным типам, мы можем выявлять нарушения договорённостей или принятых в проекте правил. Например, функция `sendRecoverLink` из предпредыдущего примера нарушает CQS:

```

async function sendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}

type RecoverPassword = (user: User) => Promise<false | void>;

// `false` – это признак запроса,
// `void` – признак команды.

```

Типы обращают внимание на подобные противоречия. Мы далее можем улучшить функцию, например, использовав в имени паттерн `try*`:

```

async function trySendRecoverLink(user: User) {
  if (!user.verified) return false;
  await api.recoverPassword(user.id);
}

type TryRecoverPassword = (user: User) => Promise<false | void>;

// Паттерн `try*` в имени явно говорит, что перед нами всё же команда,
// но она может в каком-то случае вернуть `false`.

```

Как минимум, так мы можем сделать ожидания от функции более явными. Но лучше, конечно, отделить эффект от принятия решения и отрефакторить функцию согласно CQS.

Более детально о CQS и отделении логики от эффектов мы говорили в главе о сайд-эффектах.

Дизайн API

Во время рефакторинга статическая типизация может помочь проверить, насколько понятно спроектировано API модуля или функции. Например, по сигнатуре функции мы можем проверить очевидность API, «вычеркнув» *названия* функций и аргументов:

```
function getPostContents(user: number, post: string): Promise<string> {}
// ->
function xxx(xxx: number, xxx: string): Promise<string> {}
```

Если сигнатура функции несёт мало смысла, мы можем улучшать её до тех пор, пока не начнём видеть в ней цель функции:

```
function xxx(xxx: number, xxxx: string): Promise<string> {}
// ->
function xxx(xxx: UserId, xxxx: PostSlug): Promise<string> {}
// ->
function xxx(xxx: UserId, xxxx: PostSlug): Promise<PostContents> {}

// number -> UserId: первый аргумент – ID пользователя;
// string -> PostSlug: второй аргумент – URL публикации;
// string -> PostContents: результат – содержимое публикации.

// Из сигнатуры становится понятна механика работы функции:
// запрашиваем содержимое конкретного поста по составному ключу
// из ID пользователя и URL публикации.
```

Если смысл напрашивается сам собой, значит сигнатура функции стала «говорящей». Говорящие сигнатуры несут часть контекста задачи, поэтому мы можем передать читателю в названиях функции и аргументов дополнительные знания:

```
function fetchPost(authorId: UserId, post: PostSlug): Promise<PostContents> {}

// getPostContents -> fetchPost: узнаём, что данные будут запрошены по сети;
// userId -> authorId: узнаём, как именно пользователь связан с этими постами.
```

Более подробно техника «вычёркивания» описана у Марка Симанна в книге “Code That Fits in Your Head”. ☀

Этим же правилом можно проверять, следует ли код CQS:

```

class PostReader {
    constructor(private postSource: PostStorage) {}

    getPost(id) {
        this.contents = this.postSource.fetchPost(id);
    }
}

// По сигнатуре метод `getPost` больше похож на команду, чем запрос:

type GetPost = (id: PostId) => void;

// Возможно, стоит переработать API или выбрать другое название метода:

class PostReader {
    // ...

    readPost(id: PostId): void {
        this.contents = this.postSource.fetchPost(id);
    }
}

```

К СЛОВУ

Техника «вычёркивания» более полезна для проектирования публичного API, чтобы оно было более информативным. Непубличным функциям это может быть чуть менее важно.

-
1. "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
 2. Одержанность элементарными типами, Refactoring Guru, <https://refactoring.guru/ru/smells/primitive-obsession>
 3. "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
 4. "Ubiquitous Language" by Martin Fowler, <https://martinfowler.com/bliki/UbiquitousLanguage.html>
 5. More on Functions, TypeScript Documentation, <https://www.typescriptlang.org/docs/handbook/2/functions.html>

6. Type Aliases, TypeScript Handbook, <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-aliases>
7. "Branding and Type-Tagging" by Kevin B. Greene, <https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d>
8. Factory Method, Refactoring Guru, <https://refactoring.guru/design-patterns/factory-method/typescript/example>
9. Type Compatibility, TypeScript Documentation, <https://www.typescriptlang.org/docs/handbook/type-compatibility.html>
10. Nominal & Structural Typing, Flow Documentation, <https://flow.org/en/docs/lang/nominal-structural/>
11. Совместимость типов на основе вида типизации, TypeScript: Definitive Guide, https://typescript-definitive-guide.ru/book/chapters/Sovmestimost_tipov_na_osnove_vida_tipizacii/
12. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>

Рефакторинг тестового кода

Тесты помогают рефакторить код приложения, указывая на ошибки, которые мы могли допустить. Но сами тесты — это тоже код, поэтому их также нужно держать в порядке и время от времени рефакторить.

В этой главе поговорим о том, как не поломать тесты и не снизить их надёжность во время рефакторинга и на что обращать внимание при поиске проблем с тестовым кодом.

«Тесты» для тестов

Надёжный тест падает, когда код работает не так, как ожидается. Тесты будто «прикрывают нам спину» во время изменения кода приложения, потому что ошибка в поведении отразится в тестах. Самим же тестам «прикрывает спину» продакшн-код, потому что именно в нём мы проверяем, что тесты падают по указанным причинам.

Когда тесты меняются вместе с кодом приложения, у нас не остаётся способов проверить, что всё работает *как раньше*. Обновлённые тесты могут содержать ошибку или проверять что-то *отличное от* оригинальной функциональности. Если код, по которому мы раньше могли проверить работу теста, тоже изменился, эта ошибка может остаться незамеченной.

В итоге в проекте могут появиться тесты, которым мы доверяем, но которые не работают или работают неправильно. Чтобы этого не допустить, во время рефакторинга тестового кода нам стоит следовать правилу:

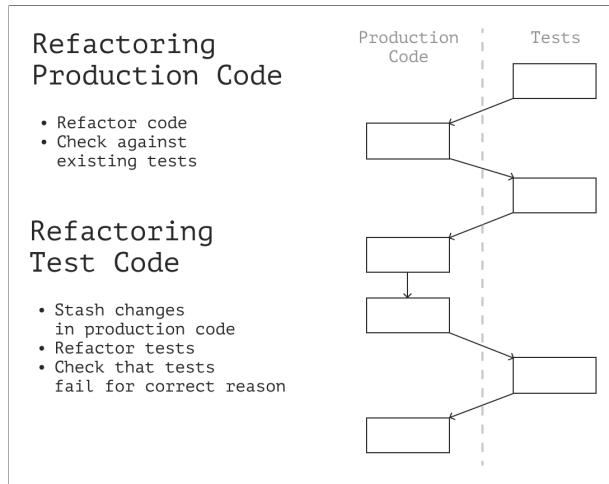
Не рефакторить тесты одновременно с кодом приложения. Лучше делать это по очереди

Если во время рефакторинга мы понимаем, что хотим отрефакторить тестовый код, то нам стоит:

- Положить на полочку изменения кода с последнего коммита (с помощью `git stash`);

- Отрефакторить код теста;
- Проверить, что он падает по указанной причине;
- Закоммитить изменения тестов;
- Достать с полочки изменения кода приложения;
- Продолжить рефакторинг.

Это правило превращает рефакторинг в «пинг-понг» между тестами и кодом приложения. Когда мы меняем код, поведение фиксируют и проверяют тесты. Когда мы меняем тесты, поведение фиксирует код приложения.



Когда мы меняем код, поведение фиксируют и проверяют тесты. Когда мы меняем тесты, поведение фиксирует код приложения

Эта техника не гарантирует, что мы не допустим никаких ошибок, но уменьшает их вероятность. При её использовании на каждом этапе есть как минимум одна часть, которая *не изменилась с последнего рабочего состояния*. Поэтому мы более уверены, что всё работает как раньше.

К СЛОВУ

Таким же образом рекомендует поступать и Марк Симанн в "Code That Fits in Your Head".⁶ Кроме самой техники в книге он также рассказывает, как не допустить ослабления тестовых критериев во время рефакторинга.

«Хрупкие» тесты

Во время работы с тестами стоит обращать внимание на свои ощущения. Если в тестах чувствуется «хрупкость» и ненадёжность, вероятно, их можно улучшить. Ощущение «дунешь, и всё развалится»

— отличный индикатор хрупких тестов.

Чаще всего хрупкими тесты делают моки. Они требовательны к внутренней структуре, порядку и способу их вызова и результату, который они должны вернуть. В запущенных случаях приложение может стать «сверх-замоканным» — когда работа почти всех модулей в тестах имитируется моками.

В таких случаях любые изменения кода приложения, даже самые незначительные, приводят к большому количеству правок в тестах. Тесты требуют больше ресурсов на поддержку, и разработка приложения замедляется. Такой эффект называют *уроном от тестов* (*test-induced damage*). ☈

В отличие от моков *стабы* и *простые тестовые данные* помогают писать более устойчивые к изменениям тесты. ☈ Стабы и тестовые данные прощают нам при их использовании гораздо больше, чем требуют. Они помогают избегать урона от тестов и тратить меньше времени на их обновление.

ПОДРОБНЕЕ

О разнице между стабами, моками и другими фейковыми объектами, хорошо написано у Microsoft. ☈

Для уменьшения хрупкости тестов мы можем руководствоваться эвристикой:

Меньше моков; проще стабы и тестовые данные

Например, бизнес-логику мы можем тестировать без моков в принципе. ☈ ☈ Это сложно, когда логика перемешана с эффектами; но проще, если она описана чистыми функциями.

Чистые функции тестируемы по своей природе. Они не требуют замороченной тестовой инфраструктуры, тесты для них могут быть запущены параллельно с разработкой. (Такие функции можно протестировать даже без тест-раннеров, сравнивая настоящий результат с желаемым — они настолько просты в тестировании.)

Посмотрим на разницу между кодом, где логика и эффекты перемешаны, и кодом, где они разделены. Обратим внимание, насколько «хрупкими» кажутся их тесты. В первом случае логика и эффекты смешаны:

```

function fetchPostList(kind) {
  const directory = path.resolve("content", kind);
  const onlyMdx = fs.readdirSync(directory).filter((f) => f.endsWith(".mdx"));
  const postNames = onlyMdx.map((f) => f.replace(/\.mdx$/, ""));
  return postNames;
}

// Для юнит-теста такой функции нам потребуется замокать `fs`,
// описать работу нужного метода,
// указать, что метод должен вернуть,
// сбросить мок после окончания теста:

it("should return a list of post names with the given kind", () => {
  jest.spyOn(fs, "readDirSync").mockImplementation(() => testFileList);
  const result = fetchPostList("blogPost");
  expect(result).toEqual(expected);
  jest.restoreAllMocks();
});

```

Во втором случае логика отделена от эффектов. Преобразование можно протестировать используя только стабы и тестовые данные:

```

function namesFromFiles(fileList) {
  return fileList
    .filter((f) => f.endsWith(".mdx"))
    .map((f) => f.replace(/\.mdx$/, ""));
}

// Для сравнения достаточно тестовых данных
// и желаемого результата работы функции:

it("should convert file list into a list of post names", () => {
  const result = namesFromFiles(testList);
  expect(result).toEqual(expected);
});

```

Структура теста становится проще, а обновление тестовых данных не отнимает большого количества ресурсов. С такой организацией кода мы даже можем полностью отказаться от статических тестовых данных и генерировать их автоматически по заранее определённым свойствам. Такое тестирование будет называться *атрибутным* (*property-based*).

Для генерации тестовых данных в property-based тестах обычно удобно использовать дополнительные инструменты.[♂] Например, в JS-экосистеме существуют библиотеки типа `faker.js`,[♂] которые создают объекты со случайными данными по заранее описанным шаблонам.

Выделенные эффекты можно протестировать отдельно интеграционными или E2E тестами. В зависимости от того, как в нашем проекте устроена работа с зависимостями типа `[fs]`, нам может быть достаточно протестировать только адаптеры к ним. Как правило, сложность и требовательность моков в таком случае будет ниже.

Например, в тестах подобного адаптера для `[fs]` нам достаточно будет проверить, что был вызван правильный метод с нужным аргументом:

TYPESCRIPT

```
function postsByType(kind) {
  const directory = path.resolve("content", kind);
  const fileList = fs.readdirSync(directory);
  return fileList;
}

// Нам уже не нужно мокать реализацию «сервиса»,
// достаточно предоставить нужное публичное API.
// Такой мок гораздо более устойчив к изменениям
// кода приложения и наносит меньше «урона от тестов».

describe("when called with a post kind", () => {
  it("should read file list from the correct directory", () => {
    const spy = jest.spyOn(fs, "readDirSync");
    postsByType("blogPost");
    expect(spy).toHaveBeenCalledWith("/content/blogPost/");
  });
});
```

Тогда сама функция `fetchPostList` превратится в «композицию» логики с эффектами:

```
function fetchPostList(kind) {
    // Чтение данных, эффект:
    const fileList = postsByType(kind);

    // Логика, чистые функции:
    return namesFromFiles(fileList);
}
```

Такую функцию проверять юнит-тестами уже может оказаться не нужно. Она объединяет функциональность разных модулей (юнитов), поэтому мы можем подумать об интеграционном или E2E-тестировании.

[ПОДРОБНЕЕ](#)

Более подробно о том, какие бывают стратегии работы с зависимостями и организации эффектов, мы говорили ранее в главах об архитектуре и сайд-эффектах.

Тесты-дубликаты

Урон от тестов замедляет разработку, потому что после каждого изменения кода приходится тратить много ресурсов на исправление тестов. Одной из причин такого замедления могут быть тесты, которые тестируют одну и ту же функциональность несколько раз.

В идеале мы хотим, чтобы за одну часть кода отвечал *один* тест. Когда тестов становится больше, мы начинаем тратить лишнее время на их обновление. Чем больше дубликатов, тем больше временной «налог».

Например, если бы в примере выше мы написали дополнительный юнит-тест для функции

`fetchPostList`, скорее всего, он бы оказался лишним и дублировал тесты функций `postsByType` и `namesFromFiles`. Тогда на каждое изменение `postsByType` или `namesFromFiles` нам бы пришлось обновлять не один тест, а два.

Тесты-дубликаты могут намекнуть на одну из нескольких проблем:

1. В коде приложения действительно может быть дублирование. Это повод провести реview и устраниТЬ повторяющуюся функциональность. (Подробнее о том, как разделять дублирование и недостаток информации о системе мы говорили в одной из предыдущих глав.)
2. В коде нечётко разделена ответственность между модулями; тесты одного модуля частично перекрывают функциональность другого. Например, один тест может проверять то, что уже проверено другими. Это может быть поводом пересмотреть стратегию тестирования и чётче определить границы модулей.

Тесты разных видов для надёжности могут перекрывать друг друга. Например, интеграционный тест может захватить часть функциональности, проверенной юнит-тестами, если так удобнее тестировать приложение.

Лично я стараюсь держать количество таких перекрываний минимальным, но в разных проектах стратегия тестирования может отличаться, поэтому дать общие рекомендации здесь сложно.

Тесты, которые никогда не ломаются

Тест должен отвечать за конкретную проблему, при появлении которой обязан упасть. Если тест никогда не падает, он вреден: пользы не приносит, но отнимает ресурсы на поддержку. Такой тест стоит удалить или переписать так, чтобы он начал падать в описанных обстоятельствах.

К СЛОВУ

Чаще всего никогда-не-падающие тесты я встречал в сверх-замоканных системах, где инфраструктура и подготовка к тесту почти полностью состояли из вызова моков. Такие тесты часто передают результат работы одного мока в другой — и в итоге не проверяют ничего.

Тесты простых функций

При выборе что и как тестировать, нам стоит сравнивать пользу от теста и его издержки. Например, можно обратить внимание на цикломатическую сложность функции, которую этот тест проверяет.

Если сложность функции равна единице, а тест приносит больше дополнительной работы, чем реальной пользы, то от теста можно отказаться.  Например, отдельный юнит-тест для функции

`fullName` может быть лишним:

JAVASCRIPT

```
const fullName = (user) => `${user.firstName} ${user.lastName}`;
```

УТОЧНЕНИЕ

Мы здесь не утверждаем, что несложным функциям тесты не нужны вовсе. Решение, тестировать или нет, зависит от конкретной ситуации. Главная идея в том, что если тест приносит больше издержек, чем пользы, стоит подумать о его необходимости.

Регрессии

Иногда простые функции всё же надо тестировать: например, если в функции когда-то была регрессия. Регрессии обращают внимание не на потенциальные, а на настоящие баги в коде, которые *действительно могут случиться и однажды случились*.

Всё, что всплыло во время регрессий, надо закрыть тестами. Если кажется, что тест слишком простой, и кто-то посчитает его бесполезным и удалит, то можно добавить аннотацию в комментарий со ссылкой на регрессию.

JAVASCRIPT

```
/**  
 * @regression JIRA-420: Users had full names in an incorrect format where last n  
 * @see https://some-project.atlassian.com/...  
 */  
describe("when called with a user object", () => {  
  it("should return a full name representation with first name at start", () => {  
    const name = fullName(42);  
    expect(name).toEqual(expected);  
  });  
});
```

-
1. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
 2. "Test-Induced Design Damage" by David Heinemeier Hansson, <https://dhh.dk/2014/test-induced-design-damage.html>
 3. "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
 4. Unit testing best practices with .NET Core and .NET Standard, Microsoft Docs, <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>
 5. Faker, Generate fake (but realistic) data for testing and development, <https://fakerjs.dev>

Рядом с кодом

Обычно рефакторинг затрагивает только код приложения и тесты. Но иногда во время рефакторинга бывает полезно коснуться комментариев в коде, проектной документации и рабочих процессов типа код-ревью.

В этой главе мы поговорим о том, как и зачем рефакторить комментарии и документацию. Обсудим, как искать противоречия друг другу источники информации о проекте и на что обращать внимание в процессах разработки.

Источники правды

Работа над проектом строится из различных задач: написания и тестирования кода, обсуждения ограничений, уточнения требований. В процессе работы над ними мы производим код, документацию, проектный план, бэклог и другое.

Для работы *приложения* из всего перечисленного реально значим только код. Он непосредственно влияет на выполнение программы, а всё остальное — нет.

К слову

Вообще, в JavaScript есть инструменты, которые используют комментарии JSDoc как сигнатуры, [♂] но это уже скорее *типы*, чем комментарии. Мы под комментариями будем иметь в виду те, что не влияют на исполнение программы.

Разработчики же учитывают не только код, но и всё, что находится «рядом с ним». Мы читаем комментарии, документацию и сообщения из код-ревью, чтобы составить более полное представление о задаче и проекте.

Обычно то, что «рядом с кодом», устаревает быстрее него. Из-за этого информация из разных источников может противоречить, а нам может быть сложно понять, чему доверять. Например, во фрагменте ниже комментарий конфликтует с сигнатурой функции:

```
/**  
 * @param {User} Current user object.  
 * @param {Product[]} List of products for the order.  
 * @return {Order}  
 */  
  
function createOrder(userId, products) {  
    /**  
     * В аннотации User, а в реализации UserId...  
     * Такое противоречие может вызвать вопрос,  
     * как должно быть на самом деле и где ошибка:  
     * - А как правильно: как работает код, или как написано в аннотации?  
     * - А что из этого менялось последним? А почему приняли такое решение?  
     * - А как написано в документации? А что скажет продукт-оунер?  
     */  
}
```

Фрагмент выше предлагает нам два противоречащих друг другу утверждения. Такие противоречия затрудняют чтение кода и замедляют разработку. Мы не можем быть уверены, что правильно поняли код, пока не разрешим их, а это может требовать усилий и времени.

Чтобы избежать подобных проблем, полезно проводить реview комментариев, документации и других источников информации о проекте, во время которых выявлять и разрешать противоречия.

Комментарии

Для «рефакторинга» комментариев мы можем воспользоваться несколькими инструментами и эвристиками:

[Лживые комментарии исправить или удалить](#)

Комментарии могут врать — то есть сообщать читателю неправильную информацию. Иногда ложь может быть откровенной:

```
/** @obsolete Not used anymore across the code base. */
function isEmpty(cart) {}

// ...

// Комментарий врёт, функция-то используется...
if (!isEmpty(cart)) {
}
```

...А иногда ложь может быть менее очевидной. Во втором случае, чтобы убедиться в лживости подозрительного комментария, мы можем попробовать опровергнуть утверждение из него.

За опровержением мы можем обратиться к другим разработчикам или документации. Например, мы можем найти коллег, которые точно знают, как функция должна работать, и спросить о подозрительном комментарии у них.

Когда возможности обратиться к команде и документации нет, мы можем провести серию экспериментов над противоречивым кодом. Нам потребуется покрыть этот код тестами и понаблюдать, как они себя ведут при изменении работы функции.

Если мы убедились, что комментарий лживый, его стоит удалить или переписать так, чтобы в нём не осталось противоречий. Этот шаг важно отметить в сообщении к коммиту в репозитории. В нём полезно написать, почему мы заподозрили, что комментарий врёт, и что именно помогло выявить ложь.

К СЛОВУ

Изменения комментариев тоже лучше оформлять в виде отдельных коммитов, как и другие техники рефакторинга. Это помогает отражать в сообщениях к коммитам контекст задачи и цель изменений.

Расплывчатые комментарии уточнить

Расплывчатым, неточным или двусмысленным комментариям стоит добавить деталей: примеров вызова функции, ссылок на конкретные PR, задачи или баги в трекере. Например, такой комментарий не очень полезен:

```
// Custom sorting function:
function sort(a, b) {}
```

То, что функция `sort` отвечает за сортировку, понятно из её названия. Вместо этого лучше описать, какой алгоритм сортировки она реализует, добавить примеров работы и ссылок на детальное

описание алгоритма:

JAVASCRIPT

```
/**  
 * Implements the most efficient sorting algorithm  
 * by comparing electron movement in the circuitry.  
 * @see https://wiki.our-project.app/sorting/electron-movement-sort/  
 * @example ...  
 *  
 * @param {Sortable} a  
 * @param {Sortable} b  
 * @return {CompareResult}  
 */  
function superFastSort(a, b) {}
```

Мелкие уточняющие комментарии перенести в код

Небольшие уточнения из комментариев можно перенести прямо в имена и сигнатуры переменных, функций или методов:

TYPESCRIPT

```
// Fetches post contents by the author's ID.  
async function getPost(user) {}  
  
// ↓  
async function fetchPostContents(authorId) {}  
  
// ↓  
async function fetchPost(authorId: UserId): Promise<PostContents> {}
```

Это срабатывает не всегда. Детали из комментария могут сделать имя переменной или функции слишком длинным. В таких случаях лучше откатить изменения.

«Пересказам названий» добавить контекста

Некоторые комментарии не несут пользы и лишь пересказывают другими словами имя сущности, которую комментируют:

JAVASCRIPT

```
// Compares strings.  
function compareString(a, b) {}
```

В таких случаях нам, опять же, будет полезнее описать в комментарии контекст задачи. Например, для функции `compareString` лучше указать, почему мы используем собственную реализацию

функции сравнения. Причина появления функции передаст больше деталей задачи и проекта в целом, чем просто пересказ имени:

JAVASCRIPT

```
/**  
 * Implements compare for our limited alphabet with custom diacritic rules.  
 * - Required for correct handling of ...  
 * - Justified as a part of R&D in ...  
 * @see https://wiki.our-project.app/sorting/custom-diacritic-comparer/  
 *  
 * @example compareString('a', 'ä') === -1  
 * @example compareString('a', 't') === -1  
 */  
function compareString(a, b) {}
```

TODO и FIXME превратить в задачи

Иногда комментарии содержат `TODO` и `FIXME` теги с описанием ошибок или недостатков в коде. Содержимое таких комментариев полезно превращать в тикеты в трекере задач проекта.

В отличие от комментариев задачи в трекере видны другим разработчикам и остальной команде. По количеству таких задач можно судить о состоянии проекта и накопившемся техническом долге.

В описании создаваемых задач стоит указать, как и почему код работает сейчас и что мы хотим изменить. Ссылки на созданные задачи можно оставить в комментарии рядом с кодом. Тогда такой комментарий:

JAVASCRIPT

```
// TODO: improve post-conditions.  
function ensureMessageSent() {}
```

...Превратится в задачу и ссылку на неё:

JAVASCRIPT

```
/**  
 * For now, it's not clear what criteria to use for the verification.  
 * Update post-conditions when the criteria are known:  
 * @see https://our-team.tasktracker.com/our-product/task-42  
 */  
function ensureMessageSent() {}
```

Наиболее эффективная стратегия для этого — проводить регулярные реview с поиском тегов и превращением их в задачи. Тогда разработчики также смогут создавать `TODO` и `FIXME` теги в

коде, чтобы «не отрываться» от работы во время написания кода. Но далее во время регулярных ревью эти комментарии будут конвертироваться в задачи, что не даст выйти количеству тегов из-под контроля.

Документация

Проектная документация хранит историю развития приложения и причины принятых технических решений. Она отвечает на вопросы разработчиков, но её проблема в том, что она устаревает быстрее кода.

Обычно документация не интегрирована в код и существует отдельно. Из-за этого обновлять её нужно дополнительно к изменениям в коде, а значит вероятность забыть об этом выше, и расхождений между ней и кодом будет становиться больше.

Обновление документации вслед за изменениями кода требует дисциплины или процесса. Чтобы документация не устаревала, стоит завести регулярную задачу на её ревью. Раз в определённый период времени (скажем, в месяц) мы будем сравнивать отличия между документацией и кодом и исправлять их.

Регулярные задачи ограничивают размер расхождений, потому что они существуют недолго и не успевают накопиться. Когда разработчики периодически их «подчищают», негативный эффект от расхождений будет меньше.

К СЛОВУ

Чтобы регулярные аудиты не превращались в чрезмерно объёмные задачи, в документации стоит хранить в основном code-agnostic информацию, которая скорее относится к предметной области и вряд ли будет меняться так же быстро, как код.

А вот архитектурно-важные решения может быть полезно хранить ближе к коду, чем остальную документацию. Если эти решения влияют на организацию кода или принципы работы приложения, разработчикам должно быть удобно к ним обращаться, не тратя лишнего времени.

Один из подходов к работе с такими решениями известен как *Architectural Decision Records*, ADR. ☈

Доступность знаний

Также полезно постараться сделать знания о проекте и предметной области как можно более доступными команде разработки. Под доступностью знаний мы будем понимать, насколько быстро и удобно разработчики могут их найти.

Чем больше разработчики знают о предметной области, тем меньше ошибок они допустят в коде приложения и больше противоречий выявят на ранних этапах жизни проекта. Чем доступнее информация в проекте, тем меньше времени будет занимать её поиск и меньше будет замедляться разработка.

Знания можно хранить по-разному: с помощью документации, метаданных репозитория, комментариев в коде или самого кода. Доступность разных вариантов отличается. Например, код доступнее всего — он всегда под рукой разработчика. Метаданные из репозитория или документация менее доступны, потому что к ним нужно обращаться отдельно.

Чтобы сделать информацию доступнее, мы можем во время регулярных реview переносить знания «ближе к коду»:

- Детали из комментариев переносить в переменные, функции и типы;
- Замечания из код-реview — в код, документацию или сообщения коммитов;
- Инсайты из «разговоров у кулера» — в документацию или трекер задач.

Эта техника может не сработать, если в проекте уже есть процесс, который регламентирует работу с документацией или репозиторием. Но в проектах «без процесса» помнить о повышении доступности информации может быть полезно.

1. JSDoc Reference, TypeScript Documentation <https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html>

2. Architectural Decision Records, ADR, <https://adr.github.io>

Рефакторинг как процесс

В прошлых главах мы уделяли основное внимание техническим деталям рефакторинга, но это только часть процесса улучшения кода в проекте.

В этой главе поговорим, как находить на рефакторинг время и заниматься им регулярно. Обсудим, как преодолевать трение в проекте и приступать к рефакторингу, даже если кодовая база большая и в ней много легаси.

Рефакторить или переписывать

Если мы работаем с легаси-кодом, то первая мысль при взгляде на него — «да проще с нуля переписать». Иногда это действительно так, но адекватно оценить ситуацию с первого взгляда можно не всегда. Перед тем как решить, рефакторить или переписывать, нам стоит оценить 3 вещи: ресурсы команды, выгоды и риски нашего решения.

Эти параметры не дадут прямого ответа на вопрос «Что делать?», но дадут более объективную оценку состояния проекта. Иногда одной такой оценки бывает достаточно, чтобы принять решение. Но даже если её не достаточно, она подскажет, каких знаний о проекте нам не хватает для принятия окончательного решения.

Ресурсы

Под *ресурсами* мы будем понимать время, знания и опыт, которые есть в распоряжении команды. Это то, что можно «тратить» на рефакторинг или переписывание кода, чтобы улучшить его.

Время

Оценивать свободное время разработчиков можно ретроспективно, смотря на прошлый опыт работы над проектом. Нам потребуется посчитать сколько времени команда уделяла улучшениям в прошлом, как часто появлялись свободные «окна» в расписании проекта.

Прошлый опыт покажет, как расходовалось время на разработку в прошлом и какие паттерны прослеживались во время неё. Это поможет спрогнозировать, сколько свободного времени у нас будет в ближайшем будущем.

Такой прогноз может показаться заниженным, потому что мы можем хотеть уделять улучшениям кода больше времени, но он отражает привычные паттерны разработки. Их полезно знать, потому что даже если мы договорились уделять больше времени рефакторингу и тех. долгу, без перемен в рабочих процессах разработка вернётся к привычным паттернам.

К тому же нам всегда стоит помнить о «непредвиденных проблемах», которые точно появятся при работе с легаси и будут отнимать на решение дополнительное время.

Накопленные знания

Накопленные знания о проекте можно оценить по количеству документации, полезных комментариев в коде, качеству истории коммитов, доступности участников, которые писали код, и выразительности самого кода.

Знания сложно оценить количественно, поэтому можно использовать качественную оценку. Чем больше противоречий между разными источниками информации мы находим, тем хуже можем считать качество накопленных знаний. И наоборот, чем стройнее выглядит модель проекта и проще найти людей, которые занимались им с самого начала, тем качество знаний выше.

Опыт

Под опытом будем иметь в виду знакомство разработчиков с текущим и различными другими проектами, языками и парадигмами. Чем разностороннее опыт, чем больше разработчики «повидали», тем меньше времени мы будем тратить на разработку нерабочих решений.

К слову

Стоит также учитывать и опыт сторонних консультантов и экспертов, если мы допускаем возможность их участия, но это сложнее сделать объективно.

Выгоды и риски

Также нам стоит определить выгоды и риски рефакторинга и переписывания кода. Они будут напрямую зависеть от рабочих процессов и «кухни» проекта, поэтому понадобится какое-то внутреннее исследование. Удобнее всего оценивать выгоды и риски параллельно, потому что стремление к любой выгоде имеет под собой какой-то риск.

Мета-информация о проекте

«Мета-информация» рассказывает о процессе работы над проектом и том, какие части кода в нём наиболее важные. Если команда использует систему контроля версий, то вся необходимая «мета-информация» уже есть там. Например, история коммитов может нам рассказать:

- что является ключевой частью проекта — в каких файлах дописывают фичи и правят баги чаще всего;
- какое неявное зацепление есть между частями проекта — какие файлы менялись вместе с другими файлами;
- в каких частях проекта было больше всего багов — какие коммиты относились к баг-фиксам и т.д.

Анализ мета-информации о проекте может рассказать, какие части кода самые сложные или самые полезные с точки зрения бизнеса.

[ПОДРОБНЕЕ](#)

Хорошо об этом написал Адам Торнхил в своей книге “Your Code as a Crime Scene”. ↗

Если системы контроля версий нет, то можно попробовать собрать косвенные показатели (релиз-ноуты, базу данных поддержки и т.д.), но делать выводы по ним будет сложнее.

Эстимейты

Если мы всё же решили отрефакторить конкретный кусок кода, то следующий шаг — спланировать итерацию.

Чтобы понять, какое количество времени может уйти на рефакторинг куска кода, сперва стоит выделить в нём места, которые вызывают вопросы. Чтобы определиться, с какими проблемами мы имеем дело, мы можем разложить проблемные места в коде по такой таблице:

	Просто	Сложно
Понятно	Ресёрч не нужен, ясно как решать, не займёт много времени	Ясно как решать, но точно замёт много времени
Непонятно	Задача маленькая и изолированная, но может потребоваться ресёрч	Точно нужен ресёрч, много скрытых связей, незнакомая область

Количество требуемого времени будет напрямую зависеть от пропорции кода в этой таблице. Чем больше сложного и непонятного, тем сложнее адекватно спланировать итерацию.

Задачи из последней ячейки спланировать сложнее всего. Для таких задач можно предложить команде использовать метод «Итерация-гипотеза». В этом методе каждое предположение о проблеме будет отдельной итерацией разработки. Оправдание или подтверждение этого предположения — цель итерации.

Проверка одной гипотезы занимает не так много времени, как исследование всей проблемы целиком. Итерации с такими проверками проще планировать и проводить. При этом с каждой проверенной гипотезой мы понимаем о проблеме больше и рано или поздно она перейдет из правой нижней ячейки в какую-то другую — тогда мы сможем точнее оценить масштаб предстоящей работы и требуемое время.

К СЛОВУ

План итерации и оценка масштаба задачи могут быть полезны, даже если команда следует философии “No Estimates.”⁹ Например, с ними легче отследить, когда пора менять стратегию работы, не успев при этом потратить лишние ресурсы на задачу.

Рефакторинг больших кусков кода

Если кусок кода не отрефакторить разом, то полезно следовать методологии «Рядом, а не вместо».

Суть подхода в том, чтобы *не заменять* кусок кода под рефакторингом, а создавать аналогичную функциональность рядом с этим кодом. Когда аналог достаточно проработан, чтобы заменить собой проблемную часть, — мы заменяем старый код на свежесозданный.

К СЛОВУ

Мне нравится аналогия этого подхода с фикусом-душителем.⁹ Фикус обвевает дерево, твердеет, а когда дерево умирает, фикус остаётся в форме этого дерева. Рефакторинг «рядом» — это такой вот фикус: мы сперва «оплетаем снаружи» фичу новым кодом, и когда он готов, убираем старый код внутри.

При работе по подходу «Рядом, а не вместо» важно часто сливать результаты в главную ветку проекта. Это поможет предотвратить конфликты в системе контроля версий и не даст процессу рефакторинга затянуться на долгое время.

Частота и гигиена

Рефакторить лучше как можно чаще. Удобнее всего рефакторить код сразу после внесения в него изменений. Идеально, если мы можем вернуться к этому коду ещё раз после отдыха, чтобы глянуть

на него свежей головой. Такое реевью помогает раньше выявить слабые решения.

При работе с легаси стоит рефакторить код до того, как фиксить в нём баги или добавлять новые фичи. После рефакторинга код будет более понятен, покрыт тестами и в целом приятнее для работы. Так мы уменьшим необходимое время на починку бага или новую фичу.

Также при разработке стоит помнить о правиле бойскаута:^{♂ ♂}

Оставлять после себя код чище, чем он был до

Метрики

Хотя рефакторинг и опирается на субъективные показатели типа красоты и читаемости, мы всё же можем использовать метрики для оценки качества внесённых изменений.

За основу мы возьмём метрики из доклада "Where does bad code come from?" и немножко расширим список.[♂] У нас получится список из 7 метрик, каждая из которых отвечает на вопрос «Сколько времени нужно, чтобы сделать XXX?»:

- **Search** — сколько времени нужно, чтобы найти требуемое место в коде.
- **Write** — чтобы написать новую фичу и покрыть её тестами.
- **Agree** — чтобы разработчики согласились, что «код хороший».
- **Read** — чтобы прочесть и понять, что кусок кода делает.
- **Modify** — чтобы изменить код под новые требования.
- **Execute** — чтобы выполнить/собрать/заплести кусок кода.
- **Debug** — чтобы найти и отладить баг.

Если мы проведём замеры этих метрик до рефакторинга и после, то сможем сравнить качество внесённых изменений. Понятно, что некоторые метрики всё ещё довольно субъективны, но их тем не менее можно выразить в цифрах. Количественное описание характеристик поможет нам заметить тренд на улучшение или ухудшение при оценке качества кода.

1. "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

2. "No Estimates" by Allen Holub, <https://youtu.be/QVBlnCTu9Ms>

3. "Strangler Fig Application" by Martin Fowler <https://martinfowler.com/bliki/StranglerFigApplication.html>

4. "Opportunistic Refactoring" by Martin Fowler <https://martinfowler.com/bliki/OpportunisticRefactoring.html>

5. "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>

6. "Where Does Bad Code Come From?" <https://youtu.be/7YpFGkG-u1w>

Заключение

Спасибо, что дочитали до конца!

Я надеюсь, техники, приёмы и ссылки из этой книжки окажутся вам полезны! Буду рад услышать ваш фидбек и дополнить книгу новыми идеями.

Ошибки, фидбек и история изменений

Опечатки и ошибки присылайте в ишью репозитория проекта или мне на почту. ↗ ↗ Я также буду рад идеям и фрагментам кода с более показательными примерами и контрпримерами описанных в книге техник.

Историю изменений и обновлений текста книги вы сможете найти в истории коммитов в репозитории. ↗

Издателям

Этот текст — большой черновик для книги о рефакторинге, которую мне бы хотелось написать в будущем.

В будущей книге мне хочется объединить разрозненные примеры кода в одно приложение, чтобы показать, как можно рефакторить не кусочки кода, а целые проекты. Мне кажется, это может быть более показательно и полезно.

Рефакторинг такого проекта охватил бы все упомянутые в этой книге приёмы, но был бы объединён общим контекстом одного приложения. Это бы позволило увидеть, как различные техники рефакторинга поддерживают друг друга и помогают работать на общий результат.

Если вам хотелось бы издать такую книгу, свяжитесь со мной по почте. ↗ Буду рад обсудить детали!

Переводчикам

Сейчас книга доступна на 2 языках: русском и английском. Если вам интересно перевести её на другие языки, пожалуйста, свяжитесь со мной по почте.[♂]

Благодарности

Спасибо Максиму Иванову (@satansdeer),[♂] Нику Лопину (@nlopin)[♂] и Артёму Самофалову (@dex157)[♂] за помощь в вычитке, факт-чекинге и ссылки на дополнительные материалы!

Также спасибо всем контрибьюторам за помощь с опечатками, формулировками и другими правками текста![♂]

Дополнительные материалы

В приложениях к этой книге вы найдёте список ссылок на книги, статьи, доклады и инструменты, которые я использую в работе сам и могу порекомендовать вам.

Также вы найдёте сокращённый список рекомендаций для поиска и исправления проблем с кодом, основанных на тезисах и доводах из предыдущих глав.

-
1. Рефакторинг на максималках, <https://github.com/bespoysarov/refactor-like-a-superhero-online-book>
 2. Почта для связи, bespoysarov@me.com
 3. Максим Иванов, <https://github.com/satansdeer>
 4. Ник Лопин, <https://github.com/nlopin>
 5. Артём Самофалов, <https://github.com/dex157>
 6. Контрибьюторы проекта, <https://github.com/bespoysarov/refactor-like-a-superhero-online-book/graphs/contributors>

Шпаргалка по техникам рефакторинга

В этом приложении собран список коротких рекомендаций для поиска и исправления проблем с кодом.

Рекомендации основаны на тезисах и доводах из предыдущих глав. Перед применением стоит прочесть текст самих глав, чтобы оценить, насколько эти приёмы подходят вашему проекту, стилю и привычкам в написании кода.

Это не список правил, которым надо неукоснительно следовать. Это скорее набор советов, которые можно принять к сведению. Используйте их осознанно.

Поиск проблем в коде

- Ищите запахи кода в проекте.
- Обращайте внимание на ощущения типа «тяжело читать, менять, тестировать, держать в голове».

Подготовка к рефакторингу

- Определите границы изменений и наметьте «швы».
- Покройте выбранную часть кода тестами.
- Опишите тестами как можно больше эдж-кейсов.
- Настройте автоматический запуск тестов и линтинг кода.
- Переведите предупреждения компилятора и линтера в разряд ошибок.

Во время рефакторинга

- Используйте маленькие шаги и атомарные коммиты.
- Создавайте компактные, но подробные пул-реквесты.
- Применяйте одну технику рефакторинга за раз.
- Чините баги и добавляйте фичи отдельно от рефакторинга.
- Соблюдайте приоритет преобразований.
- Не смешивайте рефакторинг тестов и кода приложения.
- Применяйте инструменты автоматизированного рефакторинга в IDE.
- Используйте диффи изменений для поиска ошибок.
- Проверяйте все изменения кода тестами, даже самые маленькие.

С чего проще начать

- Внедрите автоматическое форматирование кода.
- Проверьте код линтером и статическими анализаторами.
- Замените «костыли» фичами языка и окружения.

Внимание к именам

- Проясните непонятные имена.
- Расшифруйте незадокументированные аббревиатуры.
- Декомпозирийте сущности с длинными именами.
- Используйте разные имена для разных сущностей.
- Внедрите повсеместный язык.
- Исправьте врущие имена.

Работа с дублированием

- Отделите явное дублирование от недостатка информации о системе.
- Проводите регулярные аудиты дублированного кода.
- Выносите повторяющиеся данные в переменные.
- Выносите повторяющиеся действия в функции.

Абстракция как инструмент

- Описывайте намерение в названиях функций.
- Структурируйте код так, чтобы выдавать информацию читателю дозировано.
- Помните об ограничении рабочей памяти мозга, следите за количеством сущностей в функции.
- Упорядочьте информацию и используемые термины по уровням абстракции.
- Декомпозириуйте задачи на основе данных, которые в них используются.
- Проверяйте, чтобы у функции была только одна причина для изменения.
- Следите, чтобы модуль самостоятельно обеспечивал валидность своих данных.

Линейное выполнение кода

- Выделяйте состояния, через которые проходят данные приложения.
- Сделайте передачу невалидных данных в коде затруднительной.
- Валидируйте данные перед использованием.
- Используйте селекторы для «подготовки» данных под разные условия использования.
- Установите лимит на цикломатическую и когнитивную сложность кода.
- «Выпрямите» условия:
 - Используйте ранний возврат;
 - Применяйте законы де Моргана;
 - Используйте предикаты для динамических условий;
 - Используйте паттерны проектирования (Стратегия, Null-объект);
 - Применяйте паттерн-матчинг там, где возможно.

Работа с сайд-эффектами

- Чаще используйте чистые функции.
- Стремитесь к ссылочной прозрачности для более простой отладки.
- Воспринимайте данные и код по умолчанию как неизменяемые.
- Отодвигните сайд-эффекты к краям модуля / функции / юзкейса.
- Проверяйте простоту кода тестами: чем проще писать тесты, тем проще код.
- Пишите адаптеры, чтобы уменьшить зацепление и создавать меньше моков в тестах.
- Разделяйте эффекты на команды и запросы.
- Выявляйте нарушения CQS по сигнатурам и именам функций.
- Используйте разные модели для чтения и записи данных, когда это уместно.

Обработка ошибок

- Выделяйте разные виды ошибок в приложении.
- Обрабатывайте ошибки централизованно.
- Переходите к обработке ошибок как можно раньше.
- Страйтесь отобразить возможные ошибки в сигнатуре функции.
- Проверяйте входные данные перед началом работы с ними.
- Используйте логирование и аналитику в обработчиках последней надежды.
- Используйте декораторы для cross-cutting concerns.

Интеграция частей приложения

- Держите зацепление низким, а связность высокой.
- Проверяйте связность по входным, выходным данным и зависимостям кода.
- Обозначьте гарантии модулей через их публичное API.
- Сделайте общение модулей менее зацепленным:
 - Используйте сообщения / события;
 - Применяйте паттерны (Наблюдатель).
- Ограничивайте количество зависимостей модуля.
- Компонуйте преобразования данных, а не сайд-эффекты.
- Отделяйте логику от сайд-эффектов.
- Отделяйте данные от поведения.

Работа с обобщёнными алгоритмами

- Не торопитесь обобщать.
- Компонуйте сложные типы из простых вместо наследования.
- Используйте дженерики, когда уверены, что структура типа не поменяется.
- Обращайте внимание на условия, они могут указать на нарушение принципа подстановки.

Архитектура и общение с внешним миром

- Используйте повсеместный язык для моделирования домена.
- Отражайте этапы жизненного цикла данных приложения в их состояниях.
- Страйтесь не зависеть напрямую от стороннего кода.
- Добавляйте анти-коррозионный слой там, где может поменяться формат данных или API.

- Разделяйте UI-логику и бизнес-логику.
- Отделяйте конфигурацию от кода.

Статическая типизация как инструмент

- Используйте типы для описания предметной области.
- Сделайте невалидные состояния данных «невыразимыми» в типах.
- Выявляйте нарушения CQS по сигнатурам методов и функций.
- Применяйте технику «вычёркивания», чтобы определить информационную насыщенность сигнатур.

Рефакторинг тестового кода

- Рефакторьте тестовый код и код приложения по очереди.
- Используйте больше простых стабов и тестовых данных, чем сложных моков.
- Избавляйтесь от тестов-дубликатов и никогда-не-падающих тестов.

Рефакторинг документации, комментариев и процессов

- Избавьтесь от конфликтов между кодом и тем, что «рядом с ним».
- Уточните расплывчатые комментарии: добавьте контекста, примеров и причин именно такой реализации.
- Проводите регулярные аудиты комментариев, документации и задач в трекере.
- Оцените ресурсы, выгоды и риски при выборе между рефакторингом и переписыванием с нуля.
- Используйте мета-информацию о проекте из системы контроля версий для анализа состояния проекта.
- Используйте технику «Рядом, а не вместо» при рефакторинге больших кусков кода.
- Уделяйте время рефакторингу регулярно.
- Помните о правиле бойскаута.
- Опирайтесь на измеримые метрики при оценке улучшений.

Список литературы

На моё мнение о современной разработке и хорошем коде повлияли работы других людей. В этом приложении я подготовил список книг, докладов, статей, методологий и исследований, которые считаю самыми полезными и использую в работе сам.

Я разбил список по темам, схожим с темами глав. Если вас заинтересовала тема конкретной главы и вы хотите узнать больше, разделы в списке помогут вам удобнее ориентироваться среди ссылок.

Материалы могут повторяться, если относятся к нескольким темам. Я решил, что важнее собрать полный список для каждой темы, чем избавиться от дублей. Надеюсь, это окажется вам полезным.

Понятие рефакторинга и качества кода

О том, что такое рефакторинг вообще, зачем он нужен. К чему приводит неуправляемо растущая сложность проекта. Как определять «плохой» и «хороший» код.

Книги

- “The Art of Readable Code” by Dustin Boswell, Trevor Foucher,
<https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
- “Beyond Legacy Code” by David Scott Bernstein, <https://www.goodreads.com/book/show/26088456-beyond-legacy-code>
- “The Black Swan” by Nassim Nicholas Taleb,
<https://www.goodreads.com/book/show/242472.The Black Swan>
- “Clean Code” by Robert C. Martin, <https://www.goodreads.com/book/show/3735293-clean-code>
- “Refactoring” by Martin Fowler, Kent Beck,
<https://www.goodreads.com/book/show/44936.Refactoring>
- “Refactoring”, 2nd edition, by Martin Fowler, <https://www.goodreads.com/book/show/35135772-refactoring>
- “Refactoring JavaScript” by Evan Burchard, <https://www.goodreads.com/book/show/39331294-refactoring-javascript>

- “Software Design: Cognitive Aspect” by Françoise Détienne,
<https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect>
- “Working Effectively with Legacy Code” by Michael C. Feathers,
https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Видео

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, <https://youtu.be/ZsHMHuklJY>
- “Log4J & JNDI Exploit: Why So Bad?” <https://youtu.be/Opgqwn8TdlM>
- “Preventing the Collapse of Civilization” by Jonathan Blow, <https://youtu.be/pW-SOdi4Kkk>
- “Where Does Bad Code Come From?” <https://youtu.be/7YpFGKG-u1w>
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов,
<https://youtu.be/z73wmpdweQ4>

Статьи и исследования

- “Beauty Is in Simplicity”, by Jørn Ølmheim, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_05/
- Code Readability Testing, an Empirical Study,
https://www.researchgate.net/publication/299412540_Code_Readability_Testing_an_Empirical_Stud
- Evaluating Code Readability and Legibility: An Examination of Human-centric Studies,
<https://github.com/reydne/code-comprehension-review>
- “The Human Cost of Tech Debt”, by Erik Dietrich, <https://daedtech.com/human-cost-tech-debt/>
- How Readable Code Is, <https://howreadable.com>
- “Technical Debt”, DevIQ, <https://deviq.com/terms/technical-debt>
- «Попасть в окно рефакторинга» Иван Немыченко,
<https://web.archive.org/web/20201208134906/dopo.st/inem/200530110137>

Связанные понятия

- Автобусный фактор, Википедия, https://ru.wikipedia.org/wiki/Фактор_автобуса
- Закон Мёрфи, Википедия, https://ru.wikipedia.org/wiki/Закон_Мерфи
- Энтропия, Википедия, <https://ru.wikipedia.org/wiki/Энтропия>

Инструменты

- Code Smells, Refactoring Guru, <https://refactoring.guru/refactoring/smells>
- Refactoring Techniques, Refactoring Guru, <https://refactoring.guru/refactoring/techniques>

Прежде, чем начать

На что обратить внимание до начала рефакторинга. Как подготовить код к изменениям, чтобы упростить работу. Как обезопасить работу с будущими изменениями.

Книги

- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Debug It! Find, Repair, and Prevent Bugs in Your Code” by Paul Butcher,
<https://www.goodreads.com/book/show/6770868-debug-it>

- “Refactoring” by Martin Fowler, Kent Beck,
<https://www.goodreads.com/book/show/44936.Refactoring>
 - “Thinking, Fast and Slow” by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>
 - “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov,
<https://www.goodreads.com/book/show/48927138-unit-testing>
 - “Willpower Doesn’t Work” by Benjamin P. Hardy, <https://www.goodreads.com/book/show/35604684-willpower-doesn-t-work>
 - “Working Effectively with Legacy Code” by Michael C. Feathers,
https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
 - “Your Code As a Crime Scene” by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- "The Grand Unified Theory of Clean Architecture and Test Pyramid" by Guilherme Ferreira, <https://youtu.be/gHSpj2zM9Nw>
 - "How do you prepare before tackling a problem?" by Fun-fun-function, <https://youtu.be/mF-tVjXbO8Y>
 - «Изолирай это» Илья Климов, <https://youtu.be/BFhhrg6lzl8>
 - «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>

Статьи

- "Before You Refactor" by Rajith Attapattu, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_06/
 - "How to ask good questions" by Julia Evans, <https://jvns.ca/blog/good-questions/>
 - "Read Code" by Karianne Berg, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_70/
 - «Попасть в окно рефакторинга» Иван Немытченко,
<https://web.archive.org/web/20201208134906/dopo.st/inem/200530110137>
 - TDD: зачем и как, <https://bespoyasov.ru/blog/tdd-what-how-and-why/>

Связанные понятия

- Agile, гибкая методология разработки, Википедия, https://ru.wikipedia.org/wiki/Гибкая_методология_разработки
 - Оценка емкости рабочей памяти, Википедия, https://ru.wikipedia.org/wiki/Рабочая_память#Оценка_емкости_рабочей_памяти

Инструменты

- Extreme Programming, <http://www.extremeprogramming.org>
 - Test-Driven Development, TDD, <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
 - Tools for Better Thinking, <https://un/tools.co>

Во время рефакторинга

Чего избегать во время рефакторинга, как облегчить процесс. Как изолировать изменения и убедиться, что не поломался другой код. Как не выходить за рамки бюджета и держать изменения небольшими.

Книги

- "Beyond Legacy Code" by David Scott Bernstein, <https://www.goodreads.com/book/show/26088456-beyond-legacy-code>
- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Refactoring" by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936_Refactoring
- "Refactoring", 2nd edition, by Martin Fowler, <https://www.goodreads.com/book/show/35135772-refactoring>
- "Refactoring JavaScript" by Evan Burchard, <https://www.goodreads.com/book/show/39331294-refactoring-javascript>
- "Thinking, Fast and Slow" by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919_Working_Effectively_with_Legacy_Code
- «Джедайские техники» Максим Дорофеев, <https://www.goodreads.com/book/show/34656521>
- «Новые правила деловой переписки» М. Ильяков, Л. Сарычева, <https://www.goodreads.com/book/show/41070833>

Видео

- "7 Ineffective Coding Habits of Many Programmers" by Kevlin Henney, <https://youtu.be/ZsHMHuklJY>
- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- "Where Does Bad Code Come From?" <https://youtu.be/7YpFGKG-u1w>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUsUiFokr4>

Статьи

- "Convenience Is not an -ility" by Gregor Hohpe, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_19/
- "Deploy Early and Often" by Steve Berczuk, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_20/
- "How to Get Your Code Reviewed Faster" by Artem Sapegin, <https://blog.sapegin.me/all/faster-code-reviews/>
- "Use Git Tactically" by Mark Seeman, <https://stackoverflow.blog/2022/04/06/use-git-tactically/>
- "Write Better Commits, Build Better Projects" by Victoria Dye, <https://github.blog/2022-06-30-write-better-commits-build-better-projects/>
- Не надо заставлять, надо автоматизировать, <https://bespoyasov.ru/blog/do-not-push-automate-instead/>

Связанные понятия

Инструменты

- Oh Shit, Git?! <https://ohshitgit.com>
 - Transformation Priority Premise, Wikipedia
https://en.wikipedia.org/wiki/Transformation_Priority_Premise
 - Use binary search to find the commit that introduced a bug, <https://git-scm.com/docs/git-bisect>
 - Непрерывная интеграция, Википедия, https://ru.wikipedia.org/wiki/Непрерывная_интеграция
 - О системе контроля версий, <https://git-scm.com/book/ru/v2/Введение-О-системе-контроля-версий#ch01-getting-started>

Форматирование, линтинг и возможности языка

Как использовать все возможности автоматических инструментов рефакторинга и анализа кода.

Зачем знать нюансы языка и окружения, в котором исполняется код. В чём польза автоматизации.

Как консистентность помогает быстрее решать задачи.

КНИГИ

- “The Art of Readable Code” by Dustin Boswell, Trevor Foucher, <https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
 - “Autopilot: The Art & Science of Doing Nothing” by Andrew Smart, <https://www.goodreads.com/book/show/18053732-autopilot>
 - “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
 - “Refactoring JavaScript” by Evan Burchard, <https://www.goodreads.com/book/show/39331294-refactoring-javascript>
 - “Thinking, Fast and Slow” by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>

Видео

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, <https://youtu.be/ZsHMHuklIYI>
 - “Refactoring with Cognitive Complexity” by G. Ann Campbell, <https://youtu.be/eI9OKGrqU6o>
 - «ESLint & Prettier — Партизанщина» Илья Климов. <https://youtu.be/HzUfCnqzkb0>

Статьи и исследования

- “Automate Your Coding Standard” by Filip van Laenen, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_04/

- “Automatic semicolon insertion in JavaScript” by Dr. Axel Rauschmayer, <https://2ality.com/2011/05:semicolon-insertion.html>
- “Code Layout Matters” by Steve Freeman, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_13/
- How Readable Code Is, <https://howreadable.com>
- “How to Convert HTML Form Field Values to a JSON Object” by Jason Lengstorf, <https://www.learnwithjason.dev/blog/get-form-values-as-json>
- “Know Your IDE” by Heinz Kabutz, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_45/
- “Know Your Next Commit” by Dan Bergh Johnsson, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_47/
- “Use Git Tactically” by Mark Seeman, <https://stackoverflow.blog/2022/04/06/use-git-tactically/>
- “Use the Right Algorithm and Data Structure” by JC van Winkel, [https://97-things-every-programmer-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_89/](https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_89/)
- “Why robots should format our code for us”, <https://blog.sapegin.me/all/prettier/>

Инструменты

- Can I Use, support tables for web, <https://caniuse.com>
- List of EcmaScript Proposals, <https://proposals.es>
- Prettier, an opinionated code formatter, <https://prettier.io>
- Refactoring Source Code in VSCode, <https://code.visualstudio.com/docs/editor/refactoring>

Именование сущностей

Почему именование важно, как имена переменных и функций влияют на восприятие кода и скорость разработки. Почему синхронизация терминологии улучшает взаимодействие в команде. Как определить «плохие» и «хорошие» имена. Что делать с именами, которые врут.

Книги

- “Clean Code” by Robert C. Martin, <https://www.goodreads.com/book/show/3735293-clean-code>
- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Domain Modeling Made Functional” by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Refactoring” by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936_Refactoring
- “Refactoring”, 2nd edition, by Martin Fowler, <https://www.goodreads.com/book/show/35135772-refactoring>
- “Software Design: Cognitive Aspect” by Françoise Détienne, <https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect>

- "Thinking, Fast and Slow" by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Видео

- "7 Ineffective Coding Habits of Many Programmers" by Kevlin Henney, <https://youtu.be/ZsHMHuklJY>
- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- "Evolutionary Software Architectures" by Neal Ford, <https://youtu.be/CgISFhwbl3s>
- "Transforming Code into Beautiful, Idiomatic Python" <https://youtu.be/OSGv2VnC0go>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUsUiFokr4>
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>

Статьи и исследования

- "Code in the Language of the Domain" by Dan North, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_11/
- "Coding with Clarity" by Brandon Gregory, <https://alistapart.com/article/coding-with-clarity/>
- Evaluating Code Readability and Legibility: An Examination of Human-centric Studies, https://github.com/reydne/code-comprehension-review/blob/master/list_papers/AllPhasesMergedPapers-Part1.md
- "Give it five minutes" by Jason Fried, <https://signalvnoise.com/posts/3124-give-it-five-minutes>
- "How to ask good questions" by Julia Evans, <https://jvns.ca/blog/good-questions/>
- "Read Code" by Karianne Berg, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_70/
- "Stop using isLoading booleans" by Kent C. Dodds, <https://kentcdodds.com/blog/stop-using-isloading-booleans>
- Как теория близости работает в интерфейсе? <https://bureau.ru/bb/soviet/20160503/>
- Теория близости, Ководство, <https://www.artlebedev.ru/kovodstvo/sections/136/>

Связанные понятия

- Автобусный фактор, Википедия, https://ru.wikipedia.org/wiki/Фактор_автобуса
- Декларативность, Википедия, https://ru.wikipedia.org/wiki/Декларативное_программирование
- Декомпозиция, Википедия <https://ru.wikipedia.org/wiki/Декомпозиция>
- Логическая ошибка, Википедия, https://ru.wikipedia.org/wiki/Логическая_ошибка
- Уровни абстракции, Википедия, [https://ru.wikipedia.org/wiki/_Уровень_абстракции_\(программирование\)](https://ru.wikipedia.org/wiki/_Уровень_абстракции_(программирование))
- Энтропия, Википедия, <https://ru.wikipedia.org/wiki/Энтропия>

Инструменты

- Conventional Commits, <https://www.conventionalcommits.org/en/v1.0.0/>
- Naming Cheat Sheet, <https://github.com/kettanaito/naming-cheatsheet>
- Refactoring Source Code in VSCode, <https://code.visualstudio.com/docs/editor/refactoring>
- Refactoring Techniques, Refactoring Guru, <https://refactoring.guru/refactoring/techniques>
- Ubiquitous Language, <https://martinfowler.com/bliki/UbiquitousLanguage.html>

Дублирование кода

Как разделять дублирование и недостаток знаний, зачем использовать дублирование в качестве инструмента. В чём польза регулярных аудитов кода и как выработать привычку их заводить.

Книги

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, <https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
- "Refactoring" by Martin Fowler, Kent Beck, <https://www.goodreads.com/book/show/44936.Refactoring>
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- «Джедайские техники» Максим Дорофеев, <https://www.goodreads.com/book/show/34656521>

Видео

- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUsUiFokr4>
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>

Статьи

- "Beware the Share" by Udi Dahan, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_07/
- "How to ask good questions" by Julia Evans, <https://jvns.ca/blog/good-questions/>
- "The Single Responsibility Principle" by Robert C. Martin, [https://97-things-every-programmer-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_76/](https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_76/)
- "When You SHOULD Duplicate Code" by Marius Bongarts, <https://medium.com/@mariusbongarts/when-you-should-duplicate-code-b0d747bc1c67>
- «Копипаста в коде», <https://bespoyasov.ru/blog/copy-paste/>

Связанные понятия

- Don't Repeat Yourself, Wikipedia https://ru.wikipedia.org/wiki/Don't_repeat_yourself
- Разделение ответственности, Википедия, https://ru.wikipedia.org/wiki/Разделение_ответственности

Инструменты

- Copy/paste detector, jscpd, <https://www.npmjs.com/package/jscpd>
- Detect copy-pasted and structurally similar code, jsinspect, <https://github.com/danielstjules/jsinspect>
- Duplicate Code Smell, Refactoring Guru, <https://refactoring.guru/smells/duplicate-code>

Абстракция и разделение ответственности

Как и зачем использовать абстракцию. Зачем делить намерение и реализацию. Чем мешает ограничение рабочей памяти человеческого мозга. Как выдавать информацию о системе дозировано и контролируемо. Что помогает декомпозировать сложные задачи на более простые. Как удостовериться, что данные в корректном состоянии.

Книги

- "And Suddenly the Inventor Appeared: Triz, the Theory of Inventive Problem Solving" by Genrich Altshuller, https://www.goodreads.com/book/show/161916.And_Suddenly_the_InventorAppeared
- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Designing Data-Intensive Applications" by Martin Kleppmann <https://dataintensive.net>
- "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "The Humane Interface" by Jef Raskin, https://www.goodreads.com/book/show/344726.The_Humane_Interface
- "Thinking, Fast and Slow" by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>
- "Software Design: Cognitive Aspect" by Françoise Détienne, <https://www.goodreads.com/book/show/3104497-software-design-cognitive-aspect>
- "Structure and Interpretation of Computer Programs" by Harold Abelson, Gerald Jay Sussman, Julie Sussman, https://www.goodreads.com/book/show/43713.Structure_and_Interpretation_of_Computer_Programs
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- "You Don't Know JS Yet: Scope & Closures" by Kyle Simpson, <https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/scope-closures/ch8.md>
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- "7 Ineffective Coding Habits of Many Programmers" by Kevlin Henney, <https://youtu.be/ZsHMHuklIJY>
- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- "Category Theory in Life" by Eugenia Cheng, <https://youtu.be/h07oagHegNc>
- "Design, Composition, and Performance" by Rich Hickey, <https://youtu.be/MCZ3YgeEUPg>
- "Functional Design Patterns" by Scott Wlaschin, <https://youtu.be/srQt1NAHYC0>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUsUiFokr4>

Статьи

- "Climbing the infinite ladder of abstraction" by Alexis King, <https://lexi-lambda.github.io/blog/2016/08/11/climbing-the-infinite-ladder-of-abstraction/>

- “Coupling, Cohesion & Connascence” by Khalil Stemmler, <https://khalilstemmler.com/wiki/coupling-cohesion-connascence/>
- “Maintain a Single Layer of Abstraction at a Time” by Khalil Stemmler, <https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/>
- “Out of the Tar Pit”, by Ben Moseley and Peter Marks, <https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf>
- “The Power of Composition” by Scott Wlaschin, <https://fsharpforfunandprofit.com/composition/>
- “Prefer Domain-Specific Types to Primitive Types” by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/
- «Полиморфизм простыми словами» Sergey Ufocoder, <https://medium.com/devschacht/polymorphism-207d9f9cd78>
- «Декларативная валидация с помощью правила-ориентированного подхода и функционального программирования», <https://bespoyasov.ru/blog/declarative-rule-based-validation/>
- «Потерянная абстракция», <https://bespoyasov.ru/blog/missing-abstraction/>

Связанные понятия

- Абстракция, Википедия, <https://ru.wikipedia.org/wiki/Абстракция>
- Декомпозиция, Википедия <https://ru.wikipedia.org/wiki/Декомпозиция>
- Зацепление в программировании, Википедия, [https://ru.wikipedia.org/wiki/Зацепление_\(программирование\)](https://ru.wikipedia.org/wiki/Зацепление_(программирование))
- Инкапсуляция в программировании, Википедия, [https://ru.wikipedia.org/wiki/Инкапсуляция_\(программирование\)](https://ru.wikipedia.org/wiki/Инкапсуляция_(программирование))
- Оценка емкости рабочей памяти, Википедия, https://ru.wikipedia.org/wiki/Рабочая_память#Оценка_емкости_рабочей_памяти
- Разделение ответственности, Википедия, https://ru.wikipedia.org/wiki/Разделение_ответственности
- Связность в программировании, Википедия, [https://ru.wikipedia.org/wiki/Связность_\(программирование\)](https://ru.wikipedia.org/wiki/Связность_(программирование))
- Уровни абстракции, Википедия, [https://ru.wikipedia.org/wiki/Уровень_абстракции_\(программирование\)](https://ru.wikipedia.org/wiki/Уровень_абстракции_(программирование))

Инструменты

- Single Responsibility Principle, Principles of OOD, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- Tiny Types in TypeScript, <https://janmolak.com/tiny-types-in-typescript-4680177f026e>
- Tools for Better Thinking, <https://untools.co>
- What is Connascence? <https://connascence.io>
- Теория решения изобретательских задач, Википедия, https://ru.wikipedia.org/wiki/Теория_решения_изобретательских_задач

Функциональный пайплайн и линейное выполнение

Как и зачем выделять состояния данных в бизнес-процессах. В чём польза линейного выполнения кода. Как «запретить» передачу невалидных данных и обособить валидацию. В чём польза принципов функционального программирования.

Книги

- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Domain-Driven Design” by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Domain Modeling Made Functional” by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- “Refactoring JavaScript” by Evan Burchard, <https://www.goodreads.com/book/show/39331294-refactoring-javascript>
- “Thinking, Fast and Slow” by Daniel Kahneman, <https://www.goodreads.com/book/show/11468377-thinking-fast-and-slow>
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>

Видео

- “CQRS and Event Sourcing” by Greg Young, <https://youtu.be/JHGkaShoyNs>
- “Is Domain-Driven Design Overrated?” by Stefan Tilkov, <https://youtu.be/ZZp9RQEgqQ>
- “Functional architecture: The pits of success” by Mark Seemann, <https://youtu.be/US8QG9l1XW0>
- “Functional Design Patterns” by Scott Wlaschin, <https://youtu.be/srQt1NAHYC0>
- “Professor Frisby Introduces Composable Functional JavaScript” by Brian Lonsdorf, <https://egghead.io/courses/professor-frisby-introduces-composable-functional-javascript>
- “Refactoring with Cognitive Complexity” by G. Ann Campbell, <https://youtu.be/eI9OKGrqU6o>
- «Почему ваша архитектура функциональная и как с этим жить» Роман Неволин, https://youtu.be/9s_4wpzENhg
- «Быстрорасторимое проектирование» Максим Аршинов, <https://youtu.be/qJPsVvDLmQE>

Статьи

- “Apply Functional Programming Principles” by Edward Garson, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_02/
- “Constructive vs predicative data” by Hillel Wayne, <https://www.hillelwayne.com/post/constructive/>
- “Design a microservice domain model”, MSDN, <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model>
- “Designing with types: Making illegal states unrepresentable” by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/>
- “Functional design is intrinsically testable” by Mark Seemann, <https://blog.ploeh.dk/2015/05/07/functional-design-is-intrinsically-testable/>

- “Immutability: Making your code predictable” by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/correctness-immutability/>
- “Lenses in Functional Programming” by Albert Steckermeier, <https://sinusoid.es/misc/lager/lenses.pdf>
- “Making Illegal States Unrepresentable” by Hillel Wayne, <https://buttondown.email/hillelwayne/archive/making-illegal-states-unrepresentable/>
- “Parse, don’t validate” by Alexis King, <https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/>
- “The Power of Composition” by Scott Wlaschin, <https://fsharpforfunandprofit.com/composition/>

Связанные понятия

- Data Mapper, <https://martinfowler.com/eaaCatalog/dataMapper.html>
- Immutable Object, Wikipedia, https://en.wikipedia.org/wiki/Immutable_object
- Projection operations (C#), <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/projection-operations>
- Декларативность, Википедия, https://ru.wikipedia.org/wiki/Декларативное_программирование
- Композиция функций, Википедия, https://ru.wikipedia.org/wiki/Композиция_функций
- Объект передачи данных, DTO, Википедия, <https://ru.wikipedia.org/wiki/DTO>
- Функциональное программирование, Википедия, https://ru.wikipedia.org/wiki/Функциональное_программирование
- Функция-предикат, Википедия, <https://ru.wikipedia.org/wiki/Предикат>
- Чистые функции, Википедия, https://ru.wikipedia.org/wiki/Чистота_функции

Инструменты

- Bounded Context in DDD by Martin Fowler, <https://www.martinfowler.com/bliki/BoundedContext.html>
- Design Patterns, Refactoring Guru, <https://refactoring.guru/design-patterns>
- Domain Model, <https://martinfowler.com/eaaCatalog/domainModel.html>
- Specification for interoperability of common algebraic structures in JavaScript, Fantasy Land, <https://github.com/fantasyland/fantasy-land>
- Typed functional programming in TypeScript, fp-ts, <https://github.com/gcanti/fp-ts>
- The library which provides useful monads, interfaces, and lazy iterators, sweet-monads, <https://github.com/JSMonk/sweet-monads>
- Zen of Python, <https://peps.python.org/pep-0020/>

УСЛОВИЯ И СЛОЖНОСТЬ

Как организовать условия в коде, чтобы не увеличить сверх меры когнитивную нагрузку кода. Какие метрики использовать для измерения сложности. Как использовать автоматические инструменты для управления сложностью. Зачем «выпрямлять» выполнение кода и «выворачивать» условия. Как использовать законы булевой алгебры, чтобы упрощать условия. Какие шаблоны проектирования могут помочь это сделать. Как применять функциональное программирование для этих же целей.

Книги

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, <https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Debug It!: Find, Repair, and Prevent Bugs in Your Code" by Paul Butcher, <https://www.goodreads.com/book/show/6770868-debug-it>
- "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "Refactoring" by Martin Fowler, Kent Beck, <https://www.goodreads.com/book/show/44936.Refactoring>
- "Refactoring", 2nd edition, by Martin Fowler, <https://www.goodreads.com/book/show/35135772-refactoring>
- "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- "Functional architecture: The pits of success" by Mark Seemann, <https://youtu.be/US8QG9l1XW0>
- "Maybe Not" by Rich Hickey, <https://youtu.be/YRSWdGrpoug>
- "Refactoring with Cognitive Complexity" by G. Ann Campbell, <https://youtu.be/eI9OKGrqU6o>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUUiFokr4>
- «Почему ваша архитектура функциональная и как с этим жить» Роман Неволин, https://youtu.be/9s_4wpzENhg
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>

Статьи и исследования

- "Anti-if, the Missing Patterns" by Joe Wright, <https://code.joeag.com/2016/anti-if-the-missing-patterns.html>
- "Bringing Pattern Matching to TypeScript" by Gabriel Vergnaud, <https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k>
- "Cognitive Complexity. A new way of measuring understandability" by G. Ann Campbell, SonarSource SA, <https://www.sonarsource.com/docs/CognitiveComplexity.pdf>
- "A conditional sandwich example" by Mark Seemann, <https://blog.ploeh.dk/2022/02/14/a-conditional-sandwich-example/>
- "Decoupling decisions from effects" by Mark Seemann, <https://blog.ploeh.dk/2016/09/26/decoupling-decisions-from-effects/>
- "Destroy all `if`'s" by John A De Goes, <https://degoes.net/articles/destroy-all-ifs>
- "Functional design is intrinsically testable" by Mark Seemann, <https://blog.ploeh.dk/2015/05/07/functional-design-is-intrinsically-testable/>

- "Out of the Tar Pit", by Ben Moseley and Peter Marks, <https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf>
 - "Parse, don't validate" by Alexis King, <https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/>
 - "Why should you return early?" by Szymon Krajewski <https://szymonkrajewski.pl/why-should-you-return-early/>
 - «Распутываем сложные условия в коде» isqua, <https://isqua.ru/blog/2017/07/09/simplify-if-else/>
 - «Такой простой Boolean» isqua, <https://isqua.ru/blog/2016/08/14/takoi-prostoi-boolean/>
 - «Управление состоянием приложения с помощью конечного автомата»,
<https://bespoyasov.ru/blog/fsm-to-the-rescue/>

Связанные понятия

Инструменты

- `complexity`, ES Lint, <https://eslint.org/docs/latest/rules/complexity>
 - Design Patterns, Refactoring Guru, <https://refactoring.guru/design-patterns/>
 - ECMAScript Pattern Matching Proposal, <https://github.com/tc39/proposal-pattern-matching>
 - `switch-exhaustiveness-check`, ES Lint TypeScript, <https://typescript-eslint.io/rules/switch-exhaustiveness-check/>
 - `ts-pattern`, Library for Pattern Matching in TypeScript, <https://github.com/gvergnaud/ts-pattern>
 - Законы де Моргана, Википедия, https://ru.wikipedia.org/wiki/Законы_де_Моргана

Работа с сайд-эффектами

Почему сайд-эффекты делают программу сложнее и непредсказуемее. Как уменьшать количество эффектов в коде и что делать с эффектами, необходимыми для работы приложения. В чём польза чистых функций и ссылочной прозрачности. Как тестировать эффекты, зачем отделять логику от эффектов. Что такое команды и запросы, в чём смысл деления кода на них.

КНИГИ

- “Clean Architecture” by Robert C. Martin, <https://www.goodreads.com/book/show/18043011-clean-architecture>
 - “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-vour-head>

- "Domain Modeling Made Functional" by Scott Wlaschin,
<https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov,
<https://www.goodreads.com/book/show/48927138-unit-testing>
- "Working Effectively with Legacy Code" by Michael C. Feathers,
https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Видео

- "CQRS and Event Sourcing" by Greg Young, <https://youtu.be/JHGkaShoyNs>
- "Functional architecture: The pits of success" by Mark Seemann, <https://youtu.be/US8QG91XW0>
- «Быстроастроримое проектирование» Максим Аршинов, <https://youtu.be/qJPwSvDLmQE>
- «Грамотная работа с ошибками» Андрей Мелихов, <https://youtu.be/eh5flHypkDg>
- «Монады — не приговор» Виталий Брагилевский, https://youtu.be/lkXg_mjNgG4
- «Почему ваша архитектура функциональная и как с этим жить» Роман Неволин, https://youtu.be/9s_4wpzENhg

Статьи

- "Command-Query Separation" by Martin Fowler,
<https://martinfowler.com/bliki/CommandQuerySeparation.html>
- "Command-Query Responsibility Segregation" by Martin Fowler,
<https://martinfowler.com/bliki/CQRS.html>
- "CQS versus server generated IDs" by Mark Seemann, <https://blog.ploeh.dk/2014/08/11/cqs-versus-server-generated-ids/>
- "Functional architecture is Ports and Adapters" by Mark Seemann,
<https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/>
- "Functional Core in Imperative Shell" by Gary Bernhardt,
<https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell>
- "Impureim Sandwich" by Mark Seemann, <https://blog.ploeh.dk/2020/03/02/impureim-sandwich/>
- "Message Passing Leads to Better Scalability in Parallel Systems" by Russel Winder, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_57/
- "Out of the Tar Pit", by Ben Moseley and Peter Marks, <https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf>
- "A pipe operator for JavaScript" by Axel Rauschmayer, <https://2ality.com/2022/01/pipe-operator.html>
- «Разделение функций на команды и запросы», <https://bespoyasov.ru/blog/commands-and-queries/>

Связанные понятия

- Design Patterns, Refactoring Guru, <https://refactoring.guru/design-patterns>
- Immutable Object, Wikipedia, https://en.wikipedia.org/wiki/Immutable_object
- Referential Transparency, Haskell Wiki, https://wiki.haskell.org/Referential_transparency
- Чистые функции, Википедия, https://ru.wikipedia.org/wiki/Чистота_функции

Обработка ошибок

Какие бывают виды ошибок, чем они отличаются. Как и чем мешает запутанная обработка ошибок. На что обращать внимание при рефакторинге обработки ошибок в JS-коде. Какие использовать техники при наличии ограничений на технологии, парадигму или методологию.

Книги

- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "Patterns for Fault Tolerant Software" by Robert Hanmer, <https://www.goodreads.com/book/show/3346135-patterns-for-fault-tolerant-software>
- "The Pragmatic Programmer" by Andy Hunt, https://www.goodreads.com/book/show/4099.The_Pragmatic_Programmer
- "What I've Learned From Failure" by Reg Braithwaite, <https://leanpub.com/shippingsoftware/read>

Видео

- "Error handling: doing it right!" by Ruben Bridgewater, <https://youtu.be/bJ3glfA-jgo>
- "Functional Design Patterns" by Scott Wlaschin, <https://youtu.be/srQt1NAHYC0>
- "Maybe Not" by Rich Hickey, <https://youtu.be/YR5WdGrpoug>
- «Грамотная работа с ошибками» Андрей Мелихов, <https://youtu.be/eh5flHypkDg>
- «Монады — не приговор» Виталий Брагилевский, https://youtu.be/lkXg_mjNgG4
- «(Не|ну)жная монада Either на практике и в теории» Д. Махнёв, А. Кобзарь, <https://youtu.be/T6Os27MKUCQ>

Статьи

- "Against Railway-Oriented Programming" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/against-railway-oriented-programming/>
- "Dealing with Unhandled Exceptions", by Alexander Zlatkov <https://blog.sessionstack.com/how-javascript-works-exceptions-best-practices-for-synchronous-and-asynchronous-environments-39f6b59f012#ecc9>
- "Distinguish Business Exceptions from Technical" by Dan Bergh Johnssonm https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_21/
- "Don't Ignore that Error!" by Pete Goodliffe, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_26/
- "The Error Model" by Joe Duffy, <http://joeduffyblog.com/2016/02/07/the-error-model/>
- "Errors Are Not Exceptions" by swyx, <https://www.swyx.io/errors-not-exceptions>
- "Functional Error Handling with Express.js and DDD | Enterprise Node.js + TypeScript" by Khalil Stemmler, <https://khalilstemmler.com/articles/enterprise-typescript-nodejs/functional-error-handling/>
- "GraphQL error handling to the max with Typescript, codegen and fp-ts" by Ghislain Thau, <https://www.the-guild.dev/blog/graphql-error-handling-with-fp>

- “A Monad in Practicality: First-Class Failures” by Quil, <https://robotlolita.me/articles/2013/a-monad-in-practicality-first-class-failures/>
- “A mostly complete guide to error handling in JavaScript” by Valentino Gagliardi, <https://www.valentinog.com/blog/error/>
- “Notification” by Martin Fowler, <https://martinfowler.com/eaaDev/Notification.html>
- “Parse, don’t validate” by Alexis King, <https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/>
- “Railway Oriented Programming” by Scott Wlaschin, <https://fsharpforfunandprofit.com/rop/>
- “Replacing Throwing Exceptions with Notification in Validations” by Martin Fowler, <https://martinfowler.com/articles/replaceThrowWithNotification.html>
- “Stop catching errors in TypeScript; Use the Either type to make your code predictable” by Anthony Manning-Franklin, <https://antman-does-software.com/stop-catching-errors-in-typescript-use-the-either-type-to-make-your-code-predictable>
- “Using Results in TypeScript” by Dan Imhoff, <https://imhoff.blog/posts/using-results-in-typescript>
- “When life gives you lemons, write better error messages” by Jenni Nadler, <https://wix-ux.com/when-life-gives-you-lemons-write-better-error-messages-46c5223e1a2f>
- “Why should you return early?” by Szymon Krajewski <https://szymonkrajewski.pl/why-should-you-return-early/>
- «Декларативная валидация с помощью правила-ориентированного подхода и функционального программирования», <https://bespoyasov.ru/blog/declarative-rule-based-validation/>

Связанные понятия

- Cross-Cutting Concern, Wikipedia, https://en.wikipedia.org/wiki/Cross-cutting_concern
- Fail-fast, Wikipedia, <https://en.wikipedia.org/wiki/Fail-fast>

Инструменты

- Decorator Pattern, Refactoring Guru <https://refactoring.guru/design-patterns/decorator>
- `pipe`, fp-ts, <https://gcanti.github.io/fp-ts/modules/function.ts.html#pipe>
- `neverthrow`, Type-Safe Errors for JS & TypeScript, <https://github.com/supermacro/neverthrow>
- `sweet-monads`, Easy-to-use monads implementation with static types definition, <https://github.com/JSMonk/sweet-monads>
- `switch-exhaustiveness-check`, ES Lint TypeScript, <https://typescript-eslint.io/rules/switch-exhaustiveness-check/>
- `true-myth`, A library for safer and smarter error- and “nothing”-handling in TypeScript, <https://github.com/true-myth/true-myth>
- Предохранители в React, <https://ru.reactjs.org/docs/error-boundaries.html>

Интеграция модулей

Что такое зацепление и связность. Как делить приложение на модули и потом компоновать эти модули друг с другом. Зачем и по какому принципу декомпозировать задачи. В чём польза контрактов и гарантий между модулями. Как максимально расцепить модули друг от друга, но при

этом оставить возможность для их общения. В чём разница между объектной и функциональной композицией. Как работать с зависимостями. Как добиться целостности и согласованности данных.

Книги

- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Debug It!: Find, Repair, and Prevent Bugs in Your Code" by Paul Butcher, <https://www.goodreads.com/book/show/6770868-debug-it>
- "Dependency Injection in .NET" by Mark Seemann, <https://www.goodreads.com/book/show/9407722-dependency-injection-in-net>
- "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "Refactoring" by Martin Fowler, Kent Beck, https://www.goodreads.com/book/show/44936_Refactoring
- "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919_Working_Effectively_with_Legacy_Code

Видео

- "7 Ineffective Coding Habits of Many Programmers" by Kevlin Henney, <https://youtu.be/ZsHMHuklJY>
- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- "Functional architecture: The pits of success" by Mark Seemann, <https://youtu.be/US8QG91XW0>
- "Functional Design Patterns" by Scott Wlaschin, <https://youtu.be/srQt1NAHYC0>
- "Refactoring with Cognitive Complexity" by G. Ann Campbell, <https://youtu.be/el9OKGrqU6o>
- «Антипаттерны общие для всех парадигм» Тимур Шемединов, <https://youtu.be/NMUusUiFok4>
- «Контрактное программирование» Тимур Шемединов, https://youtu.be/k5_ksUvbGEO.
- «Почему ваша архитектура функциональная и как с этим жить» Роман Неволин, https://youtu.be/9s_4wpzENhg

Статьи

- "Dependency Injection Using the Reader Monad" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/dependencies-3/>
- "Dependency Interpretation" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/dependencies-4/>
- "Dependency Rejection" by Mark Seemann, <https://blog.ploeh.dk/2017/02/02/dependency-rejection/>
- "Evans Classification" by Martin Fowler, <https://martinfowler.com/bliki/EvansClassification.html>
- "Inversion of Control Containers and the Dependency Injection pattern" by Martin Fowler, <https://martinfowler.com/articles/injection.html>
- "Six approaches to dependency injection" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/dependencies/>
- Внедрение зависимостей с TypeScript на практике, <https://bespoyasov.ru/blog/di-ts-in-practice/>

Связанные понятия

- Design By Contract, <https://wiki.c2.com/?DesignByContract>
- Message Broker, [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648849(v=pandp.10))
- Message Bus, [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff647328(v=pandp.10))
- Message Queue, Wikipedia, https://en.wikipedia.org/wiki/Message_queue
- Заселение в программировании, Википедия, [https://ru.wikipedia.org/wiki/Заселение_\(программирование\)](https://ru.wikipedia.org/wiki/Заселение_(программирование))
- Разделение ответственности, Википедия, https://ru.wikipedia.org/wiki/Разделение_ответственности
- Связность в программировании, Википедия, [https://ru.wikipedia.org/wiki/Связность_\(программирование\)](https://ru.wikipedia.org/wiki/Связность_(программирование))

Инструменты

- Design Patterns, Refactoring Guru, <https://refactoring.guru/design-patterns/>
- React software design patterns, <https://github.com/themithy/react-design-patterns>
- Reactive Extensions Library for JavaScript, RxJS, <https://rxjs.dev>
- REpresentational State Transfer, REST, <https://restfulapi.net>
- What is Connascence? <https://connascence.io>

Обобщения и иерархии

Как понять, когда нужен обобщённый алгоритм или тип. Почему в большей части случаев композиция предпочтительнее наследования. Как использовать принцип подстановки Лисков в качестве интеграционного линтера.

Видео

- “Category Theory in Life” by Eugenia Cheng, <https://youtu.be/ho7oagHegNc>
- “The Grand Unified Theory of Clean Architecture and Test Pyramid” by Guilherme Ferreira, <https://youtu.be/gHSpi2zM9Nw>
- “Functional Design Patterns” by Scott Wlaschin, <https://youtu.be/srQt1NAHYC0>
- «Контрактное программирование» Тимур Шемсединов, https://youtu.be/k5_kSUvbGEQ.
- «Метапрограммирование и мультипарадигменное программирование» Тимур Шемсединов, <https://youtu.be/Bo9y4IxdNRY>

Статьи

- “A behavioral notion of subtyping” by Barbara H. Liskov, Jeannette M. Wing, <https://dl.acm.org/doi/10.1145/197320.197383>
- “Coupling, Cohesion & Connascence” by Khalil Stemmler, <https://khalilstemmler.com/wiki/coupling-cohesion-connascence/>
- “The Ins and Outs of Generic Variance in Kotlin” by Dave Leeds, <https://typealias.com/guides/ins-and-outs-of-generic-variance/>
- “Maintain a Single Layer of Abstraction at a Time” by Khalil Stemmler, <https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/>
- “The Power of Composition” by Scott Wlaschin, <https://fsharpforfunandprofit.com/composition/>

- “Why variance matters” by Ted Kaminski <https://www.tedinski.com/2018/06/26/variance.html>

Связанные понятия

- Covariance and Contravariance, Wikipedia,
[https://en.wikipedia.org/wiki/Covariance_and_contravariance_\(computer_science\)](https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)).
- Design By Contract, <https://wiki.c2.com/?DesignByContract>
- Generics in TypeScript, <https://www.typescriptlang.org/docs/handbook/2/generics.html>
- The Liskov Substitution Principle,
<https://web.archive.org/web/2015112804108/http://www.objectmentor.com/resources/articles/lsp.pdf>
- Types and Typeclasses, <http://learnyouahaskell.com/types-and-typeclasses>

Архитектура и ограничения

Чем может мешать слабая архитектура при рефакторинге. Как использовать повсеместный язык для улучшения архитектуры. Как строить взаимодействие с внешним миром и работу с зависимостями. В чём польза портов и адаптеров. Зачем разделять UI-логику от бизнес-логики. Как архитектура влияет на тестируемость.

Книги

- “Clean Architecture” by Robert C. Martin, <https://www.goodreads.com/book/show/18043011-clean-architecture>
- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Designing Data-Intensive Applications” by Martin Kleppmann <https://dataintensive.net>
- “Domain-Driven Design” by Eric Evans,
https://www.goodreads.com/book/show/179133.Domain_Driven_Design
- “Domain Modeling Made Functional” by Scott Wlaschin,
<https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- “Enterprise Integration Patterns” by Gregor Hohpe,
https://www.goodreads.com/book/show/85012_Enterprise_Integration_Patterns
- “Software Architecture in Practice” by L. Bass, P. Clements, R. Kazman,
https://www.goodreads.com/book/show/70143_Software_Architecture_in_Practice
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov,
<https://www.goodreads.com/book/show/48927138-unit-testing>
- “What I've Learned From Failure” by Reg Braithwaite, <https://leanpub.com/shippingsoftware/read>
- “Your Code As a Crime Scene” by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- “CQRS and Event Sourcing” by Greg Young, <https://youtu.be/IHGkaShoyNs>
- “Design, Composition, and Performance” by Rich Hickey, <https://youtu.be/MCZ3YgeEUPg>
- “Evolutionary Software Architectures” by Neal Ford, <https://youtu.be/CgISFhwbl3s>
- “Is Domain-Driven Design Overrated?” by Stefan Tilkov, <https://youtu.be/ZZp9QEGeqQ>

- “The Grand Unified Theory of Clean Architecture and Test Pyramid” by Guilherme Ferreira, <https://youtu.be/gHSpi2zM9Nw>
- “Functional architecture: The pits of success” by Mark Seemann, <https://youtu.be/US8QG9I1XW0>
- “Functional Design Patterns” by Scott Wlaschin, <https://youtu.be/srOt1NAHYC0>
- «Почему ваша архитектура функциональная и как с этим жить» Роман Неволин, https://youtu.be/9s_4wpzENhg
- «Быстростворимое проектирование» Максим Аршинов, <https://youtu.be/gjPwSvDLmOE>

Статьи

- “DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together” by Herberto Graça, <https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-put-it-all-together/>
- “Functional Architecture is Ports and Adapters” by Mark Seemann, <https://blog.ploeh.dk/2016/03/18/functional-architecture-is-ports-and-adapters/>
- “Functional Core in Imperative Shell” by Gary Bernhardt, <https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell>
- “How to ask good questions” by Julia Evans, <https://jvns.ca/blog/good-questions/>
- “Impureim Sandwich” by Mark Seemann, <https://blog.ploeh.dk/2020/03/02/impureim-sandwich/>
- “The pedantic checklist for changing your data model in a web application” by Raphael Gaschignard, <https://rtppg.co/2021/06/07/changes-checklist.html>
- “Ports & Adapters Architecture” by Herberto Graça, <https://herbertograca.com/2017/09/14/ports-adapters-architecture/>
- “The Software Architecture Chronicles” by Herberto Graça, <https://herbertograca.com/2017/07/03/the-software-architecture-chronicles/>
- “Test-Induced Design Damage” by David Heinemeier Hansson, <https://dhh.dk/2014/test-induced-design-damage.html>
- “View Models”, Reactive UI, <https://www.reactiveui.net/docs/handbook/view-models/>
- «System Design для амых маленьких», Виктор Карпов, <https://github.com/vitkarpov/coding-interviews-blog-archive/blob/main/posts/what-is-system-design.md>

Связанные понятия

- List of System Quality Attributes, Wikipedia, https://en.wikipedia.org/wiki/List_of_system_quality_attributes
- Software Requirements, Wikipedia, https://en.wikipedia.org/wiki/Software_requirements
- Ubiquitous Language, <https://martinfowler.com/bliki/UbiquitousLanguage.html>
- Реактивное программирование, Википедия, https://ru.wikipedia.org/wiki/Реактивное_программирование
- Model-View-ViewModel, Википедия, <https://ru.wikipedia.org/wiki/Model-View-ViewModel>

Инструменты

- Anti-corruption Layer Pattern, <https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer>
- Every Programmer Should Know, Architecture, <https://github.com/mtdvio/every-programmer-should-know#architecture>
- Feature-Sliced Design, Architectural methodology for frontend projects, <https://feature-sliced.design>
- Ubiquitous Language, <https://martinfowler.com/bliki/UbiquitousLanguage.html>

Декларативность

В чём польза декларативного стиля кода по сравнению с императивным. В каких ситуациях наоборот предпочтеть императивный стиль. Зачем могут пригодиться конечные автоматы во фронтенд-разработке.

Книги

- "Antifragile: Things That Gain from Disorder" by Nassim Nicholas Taleb, <https://www.goodreads.com/book/show/13530973-antifragile>
- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, <https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- "Design, Composition, and Performance" by Rich Hickey, <https://youtu.be/MCZ3YgeEUPg>
- "Designing Declarative APIs" by Ilya Birman, <https://youtu.be/uCQ3jFuQ7bQ>
- "Goodbye, useEffect" by David Khorshid, <https://youtu.be/HPoC-k7Rxwo>
- "Refactoring with Cognitive Complexity" by G. Ann Campbell, <https://youtu.be/eI9OKGrqU6o>
- "Type Level Programming in TypeScript" by Gabriel Vergnaud, <https://youtu.be/vGVvJuazs84>
- «Метапрограммирование и мультипарадигменное программирование» Тимур Шемсединов, <https://youtu.be/Bo9y4IxdNRY>

Статьи

- "Simplicity Comes from Reduction" by Paul W. Homer, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_75/
- "Maintain a Single Layer of Abstraction at a Time" by Khalil Stemmler, <https://khalilstemmler.com/articles/oop-design-principles/maintain-a-single-layer-of-abstraction/>
- "Prefer Domain-Specific Types to Primitive Types" by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/
- «Декларативная валидация с помощью правила-ориентированного подхода и функционального программирования», <https://bespoyasov.ru/blog/declarative-rule-based-validation/>
- «Управление состоянием приложения с помощью конечного автомата», <https://bespoyasov.ru/blog/fsm-to-the-rescue/>

Связанные понятия

- Transformation Priority Premise, Wikipedia
https://en.wikipedia.org/wiki/Transformation_Priority_Premise
- Конечный Автомат, Википедия, https://ru.wikipedia.org/wiki/Конечный_автомат
- Сопоставление с образцом, Википедия, https://ru.wikipedia.org/wiki/Сопоставление_с_образцом

Инструменты

- 12 Factor Apps, <https://12factor.net>
- JavaScript and TypeScript finite state machines and statecharts, XState, <https://github.com/statelyai/xstate>
- Zen of Python, <https://peps.python.org/pep-0020/>

Статическая типизация

Как использовать типы для передачи дополнительных данных о предметной области. Как сделать невалидные состояния данных невыразимыми в коде. Как использовать типы для проверки кода на нарушение договорённостей и принципов разработки, принятых в команде.

Книги

- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Domain Modeling Made Functional" by Scott Wlaschin, <https://www.goodreads.com/book/show/34921689-domain-modeling-made-functional>
- "Domain-Driven Design" by Eric Evans, https://www.goodreads.com/book/show/179133.Domain_Driven_Design

Видео

- "Category Theory in Life" by Eugenia Cheng, <https://youtu.be/ho7oagHeqNc>
- "Type Level Programming in TypeScript" by Gabriel Vergnaud, <https://youtu.be/vGVvluazs84>

Статьи

- "Branding and Type-Tagging" by Kevin B. Greene, <https://medium.com/@KevinBGreene/surviving-the-typescript-ecosystem-branding-and-type-tagging-6cf6e516523d>
- "Bringing Pattern Matching to TypeScript" by Gabriel Vergnaud, <https://dev.to/gvergnaud/bringing-pattern-matching-to-typescript-introducing-ts-pattern-v3-0-o1k>
- "Code in the Language of the Domain" by Dan North, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_11/
- "Constructive vs predicative data" by Hillel Wayne, <https://www.hillelwayne.com/post/constructive/>
- "Designing with types: Making illegal states unrepresentable" by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/>
- "Making Illegal States Unrepresentable" by Hillel Wayne, <https://buttondown.email/hillelwayne/archive/making-illegal-states-unrepresentable/>
- "Prefer Domain-Specific Types to Primitive Types" by Einar Landre, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_65/

Связанные понятия

- Nominal & Structural Typing, Flow Documentation, <https://flow.org/en/docs/lang/nominal-structural/>
- Ubiquitous Language, <https://martinfowler.com/bliki/UbiquitousLanguage.html>
- Type Aliases, TypeScript Handbook, <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#type-aliases>

- Type Compatibility, TypeScript Documentation, <https://www.typescriptlang.org/docs/handbook/type-compatibility.html>
- Совместимость типов на основе вида типизации, TypeScript: Definitive Guide, https://typescript-definitive-guide.ru/book/chapters/Sovmestimost_tipov_na_osnove_vida_tipizaci/

Инструменты

- Tiny Types in TypeScript, <https://janmolak.com/tiny-types-in-typescript-4680177f026e>
- **switch-exhaustiveness-check**, ESLint TypeScript, <https://typescript-eslint.io/rules/switch-exhaustiveness-check/>

Рефакторинг тестового кода

Как не сломать тесты во время рефакторинга, чем проверять тесты после него. Что делает тесты «хрупкими». Как найти баланс между высоким покрытием и низким «уроном от тестов».

Книги

- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Unit Testing: Principles, Practices, and Patterns” by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
- “Working Effectively with Legacy Code” by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code

Видео

- “7 Ineffective Coding Habits of Many Programmers” by Kevlin Henney, <https://youtu.be/ZsHMHukIJY>
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>

Статьи

- “Choosing properties for property-based testing” by Scott Wlaschin, <https://fsharpforfunandprofit.com/posts/property-based-testing-2/>
- “Test-Induced Design Damage” by David Heinemeier Hansson, <https://dhh.dk/2014/test-induced-design-damage.html>
- “Unit testing best practices with .NET Core and .NET Standard”, MSDN, <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>
- «TDD: зачем и как», <https://bespoyasov.ru/blog/tdd-what-how-and-why/>

Инструменты

- Faker, Generate fake (but realistic) data for testing and development, <https://fakerjs.dev>
- Test-Driven Development, TDD, <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
- **git stash**, Stash the changes in a dirty working directory away, <https://git-scm.com/docs/git-stash>

Комментарии, документация и процессы

Как синхронизировать источники правды в проекте и предотвратить их рассинхронизацию. Как и зачем делать комментарии информационно насыщеннее. Как увеличить пользу документации, не увеличив затраты на её ведение. Чему в проекте уделять внимание «кроме кода».

Книги

- “Beyond Legacy Code” by David Scott Bernstein, <https://www.goodreads.com/book/show/26088456-beyond-legacy-code>
- “Code That Fits in Your Head” by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- “Envisioning Information” by Edward R. Tufte, https://www.goodreads.com/book/show/17745_Envisioning_Information
- “The Newspaper Designer’s Handbook” by Tim Harrower, <https://www.goodreads.com/book/show/61881153-the-newspaper-designer-s-handbook>
- «Справочник издателя и автора» Аркадий Мильчин, <https://www.goodreads.com/book/show/13611575>

Статьи и исследования

- “How to ask good questions” by Julia Evans, <https://jvns.ca/blog/good-questions/>
- “Information Density and Linguistic Encoding”, by Matthew W. Crocker, Vera Demberg, Elke Teich https://www.researchgate.net/publication/283938800_Information_Density_and_Linguistic_Encoding IDEal
- “Use Git Tactically” by Mark Seeman, <https://stackoverflow.blog/2022/04/06/use-git-tactically/>
- “When life gives you lemons, write better error messages” by Jenni Nadler, <https://wix-ux.com/when-life-gives-you-lemons-write-better-error-messages-46c5223e1a2f>
- “Write Better Commits, Build Better Projects” by Victoria Dye, <https://github.blog/2022-06-30-write-better-commits-build-better-projects/>
- «Копипаста в коде», <https://bespoysarov.ru/blog/copy-paste/>

Инструменты

- Architectural Decision Records, ADR, <https://adr.github.io>
- Every Programmer Should Know Practices, <https://github.com/mtdvio/every-programmer-should-know#practices>
- JSDoc Reference, TypeScript Documentation <https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html>

Частота, процессы и метрики

Как понять, рефакторить или переписывать код. Какую информацию о проекте собрать перед началом работы. Чем руководствоваться при расчёте времени на задачу. Какими метриками измерять влияние рефакторинга на код.

Книги

- "The Art of Readable Code" by Dustin Boswell, Trevor Foucher, <https://www.goodreads.com/book/show/8677004-the-art-of-readable-code>
- "Beyond Legacy Code" by David Scott Bernstein, <https://www.goodreads.com/book/show/26088456-beyond-legacy-code>
- "Code That Fits in Your Head" by Mark Seemann, <https://www.goodreads.com/book/show/57345272-code-that-fits-in-your-head>
- "Refactoring" by Martin Fowler, Kent Beck, <https://www.goodreads.com/book/show/44936.Refactoring>
- "Unit Testing: Principles, Practices, and Patterns" by Vladimir Khorikov, <https://www.goodreads.com/book/show/48927138-unit-testing>
- "Working Effectively with Legacy Code" by Michael C. Feathers, https://www.goodreads.com/book/show/44919.Working_Effectively_with_Legacy_Code
- "Write Better Commits, Build Better Projects" by Victoria Dye, <https://github.blog/2022-06-30-write-better-commits-build-better-projects/>
- "Your Code As a Crime Scene" by Adam Tornhill, <https://www.goodreads.com/book/show/23627482-your-code-as-a-crime-scene>

Видео

- "All the Little Things" by Sandi Metz, <https://youtu.be/8bZh5LMaSmE>
- "Evolutionary Software Architectures" by Neal Ford, <https://youtu.be/CgISFhwbl3s>
- "No Estimates" by Allen Holub, <https://youtu.be/QVBInCTu9Ms>
- "Preventing the Collapse of Civilization" by Jonathan Blow, <https://youtu.be/pW-SOdj4Kkk>
- "Where Does Bad Code Come From?" <https://youtu.be/7YpFGkG-u1w>
- «Рефакторинг: причины, цели, техники и процесс» Тимур Шемсединов, <https://youtu.be/z73wmpdweQ4>
- «Антипаттерны общие для всех парадигм» Тимур Шемсединов, <https://youtu.be/NMUsUiFokr4>

Статьи

- "The Boy Scout Rule" by Robert C. Martin, https://97-things-every-x-should-know.gitbooks.io/97-things-every-programmer-should-know/content/en/thing_08/
- "Strangler Fig Application" by Martin Fowler <https://martinfowler.com/bliki/StranglerFigApplication.html>
- "Opportunistic Refactoring" by Martin Fowler <https://martinfowler.com/bliki/OpportunisticRefactoring.html>
- "When to Refactor" by Mark Seemann, <https://blog.ploeh.dk/2022/09/19/when-to-refactor/>

Инструменты

- Every Programmer Should Know, Engineering Philosophy, <https://github.com/mtdvio/every-programmer-should-know#engineering-philosophy>
- Every Programmer Should Know, Practices, <https://github.com/mtdvio/every-programmer-should-know#practices>