ГЛАВА 1

Чистый код

Мартин Фаулер (Martin Fowler) в своей книге «Refactoring: Improving the Design of Existing Code»[1] (https://learning.oreilly.com/library/view/refactoring-improving-the/9780134757681) раскрыл силу и преимущества рефакторинга, а также причины, по которым он необходим. К счастью, теперь, спустя более двух десятков лет, большинство разработчиков прекрасно знают, что такое рефакторинг и «запах» кода. Они ежедневно сталкиваются с техническим долгом, а рефакторинг стал неотъемлемой частью разработки ПО. В своей основополагающей работе Фаулер использовал рефакторинг для устранения запахов кода. В этой книге мы рассмотрим это применение в виде семантических рецептов, призванных улучшить ваши решения.

1.1. Что такое запах кода?

Запах кода — это признак проблемы. Люди склонны думать, что наличие запаха говорит о том, что весь объект нужно полностью пересмотреть и переделать. Но это противоречит изначальному определению. Запах кода — это просто индикатор возможности улучшения. Он не всегда подскажет, что конкретно не так; он просто сигнализирует, что на код стоит обратить особое внимание.

Рецепты, предложенные в этой книге, представляют собой решения определенных проблем. Как и в любой книге рецептов, они не являются чем-то неизменным и обязательным, а запахи кода — это рекомендации и эвристика, а не жесткие правила. Прежде чем слепо применять любой из рецептов, вы должны разобраться в проблеме и оценить преимущества и недостатки собственного дизайна и кода. Хороший дизайн предполагает баланс между рекомендациями и практическими и контекстуальными соображениями.

1.2. Что такое рефакторинг?

Вернемся к книге Мартина Фаулера, в которой даны два взаимодополняющих определения:

[1] Фаулер М. «Рефакторинг: улучшение существующего кода».

Рефакторинг (как операция): изменение, которому подверглась внутренняя структура программного обеспечения с целью упрощения его понимания и удешевления его модификации без изменения его внешнего поведения.

Рефакторинг (как процесс): реструктуризация программного обеспечения путем применения серии операций рефакторинга без изменения его наблюдаемого поведения.

Понятие «рефакторинг» ввел Уильям Опдайк (William Opdyke) в своей диссертации на соискание степени PhD «Рефакторинг объектно-ориентированных фреймворков» (Refactoring Object-Oriented Frameworks) (https://oreil.ly/zBCkI) 1992 года, а популярным оно стало после выхода книги Фаулера. Со времени своего описания Фаулером рефакторинг сильно эволюционировал. Большинство современных интегрированных сред разработки (IDE) поддерживают автоматический рефакторинг.

Такой рефакторинг безопасен и позволяет вносить структурные изменения, не меняя поведения системы. В данной книге представлено множество рецептов автоматического, безопасного рефакторинга, а также семантического рефакторинга. Семантический рефакторинг небезопасен, поскольку он может частично изменить поведение системы. Рецепты семантического рефакторинга следует применять осторожно, поскольку они могут нарушить работу программы. Я буду указывать, есть ли в рецепте семантический рефакторинг. Если у вас хорошее покрытие поведенческого кода, вы можете быть уверены, что не нарушите важные бизнес-сценарии. При этом не стоит применять рецепты рефакторинга одновременно с исправлением дефекта или разработкой новой функции.

Большинство современных организаций имеют в своих пайплайнах непрерывной интеграции/непрерывной доставки надежные тестовые наборы. См.: Титус Уинтерс (Titus Winters) и др. «Software Engineering at Google», O'Reilly[1], 2020 (https://learning.oreilly.com/library/view/software-engineering-at/9781492082781), чтобы узнать, есть ли у вас эти пакеты тестов.

1.3. Что такое рецепт

Я использую термин «рецепт» аккуратно. Рецепт — это набор инструкций по созданию или изменению чего-либо. Рецепты, приведенные в данной книге, будут наиболее полезны, если вы разберетесь в их сути и сможете использовать их по своему усмотрению. Другие книги рецептов из этой серии более конкретны и содержат пошаговые решения. Чтобы воспользоваться рецептами из моей книги, вам нужно будет привести их в соответствие со своим языком программирования

[1] Уинтерс Т., Маншрек Т., Райт Х. «Делай как в Google. Разработка программного обеспечения». — СПб., издательство «Питер».

и проектным решением. В данном случае «рецепт» — это средство обучения, которое научит вас, как распознать проблему, определить последствия и усовершенствовать код.

1.4. Зачем нужен чистый код

Чистый код легко читать, понимать и поддерживать. Он хорошо структурирован, краток и использует осмысленные имена для переменных, функций и классов. Он создан в соответствии с лучшими практиками и паттернами проектирования и в первую очередь с вниманием к читаемости и поведению, а не к производительности и деталям реализации.

Чистый код очень важен для всех развивающихся систем, в которые вы ежедневно вносите изменения. Он особенно актуален в некоторых средах, где невозможно внедрить обновления так быстро, как хотелось бы. К ним относятся встраиваемые системы, космические зонды, смарт-контракты, мобильные и многие другие приложения.

Классические книги по рефакторингу, веб-сайты и IDE в основном посвящены рефакторингу, который не меняет поведения системы. В этой книге есть несколько рецептов для подобных сценариев, например безопасное переименование. Но вы также найдете несколько рецептов, связанных с семантическим рефакторингом, позволяющим изменить способ решения некоторых проблем. Разберитесь в коде, проблеме и рецепте, чтобы внести соответствующие изменения.

1.5. Читаемость, производительность или и то и другое

Эта книга посвящена чистому коду. Некоторые из ее рецептов не самые производительные. При возникновении такого рода противоречий я обычно делаю выбор в пользу читаемости, а не производительности. Например, я посвятил целую главу (главу 16) преждевременной оптимизации, которая не гарантирует решения проблем с производительностью.

Для критически важных по производительности решений лучшая стратегия — написать чистый код, обеспечить тестовое покрытие, а затем устранить узкие места, используя правила Парето. Принцип Парето применительно к программному обеспечению гласит, что, устранив 20 % критических узких мест, вы улучшите производительность продукта на 80 %. Если вы улучшите производительность на 20 %, это, скорее всего, увеличит скорость работы системы на 80 %.

Этот принцип защищает от проведения преждевременной оптимизации без обоснованных сценариев, поскольку она приведет лишь к незначительному улучшению и нарушит чистоту кода.

1.6. Типы программ

Большинство рецептов, приведенных в этой книге, ориентированы на бэкенд-системы со сложными бизнес-правилами. Симулятор, который вы будете создавать, начиная с главы 2, идеально подходит для этого. Поскольку рецепты не зависят от предметной области, вы также можете использовать большинство из них для разработки фронтенда, баз данных, встраиваемых систем, блокчейна и многих других сценариев. Есть также специальные рецепты с примерами кода для UX, фронтенда, смарт-контрактов и других специфических областей (например, см. рецепт 22.7 «Как скрыть низкоуровневые ошибки от конечных пользователей»).

1.7. Машинно-генерируемый код

Нужен ли чистый код сейчас, когда появилось множество инструментов для автоматической генерации кода? В 2023 году ответ — да. Даже больше, чем когда-либо. Существует множество коммерческих программных ассистентов для написания кода. Однако они не обладают (пока) полным контролем; они — вторые пилоты и помощники, а проектные решения по-прежнему принимают люди.

На момент написания этой книги большинство коммерческих инструментов и инструментов искусственного интеллекта могли создавать только анемичные решения и стандартные алгоритмы. Но они чрезвычайно полезны, если вы не помните, как написать маленькую функцию, и удобны для перевода с одного языка программирования на другой. Я активно использовал их при работе над этой книгой. Я не изучал все эти 25+ языков, которые использовал в рецептах. Я просто перевел и протестировал несколько фрагментов кода на разных языках с помощью вспомогательных инструментов. Я предлагаю вам также использовать все доступные средства, чтобы перевести рецепты из этой книги на привычный для вас язык. Инструменты никуда не денутся. Будущие разработчики станут технологическими кентаврами: наполовину людьми, наполовину машинами.

1.8. Используемые термины

На протяжении всей книги я использую следующие термины как взаимозаменяемые:

- Методы/функции/процедуры.
- Атрибуты/переменные экземпляра/свойства.
- Протокол/поведение/интерфейс.
- Аргументы/взаимодействующие объекты/параметры.
- Анонимные функции/замыкания/лямбда-выражения.

Различия между ними тонкие и иногда зависят от языка. Я буду добавлять примечания, когда необходимо будет уточнить использование того или иного термина.

1.9. Паттерны проектирования

Эта книга предполагает, что читатель имеет базовое представление о концепциях объектно-ориентированного проектирования. Некоторые рецепты основаны на популярных паттернах проектирования, включая те, что описаны в книге так называемой «Банды четырех» «Design Patterns» (https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/)[1]. Другие описывают менее известные паттерны, такие как *нулевой объект* (null object) и *объект методов* (method object). Кроме того, книга содержит объяснения и рекомендации по замене паттернов, которые теперь считаются антипаттернами, например паттерн «*Одиночка*» (Singleton) из рецепта 17.2 «Замена паттернов “Одиночка”».

1.10. Парадигмы языков программирования

По мнению Дэвида Фарли (David Farley) (https://learning.oreilly.com/library/view/modern-software-engineering/9780137314942/ch02.xhtml#ch2lev4)[2],

> одержимость нашей отрасли языками и инструментами вредит нашей профессии. Это не означает, что в проектировании языков не было достижений, однако большая часть усилий при разработке языков, была сосредоточена, например, на синтаксисе, а не на структурных особенностях, что неверно.

Концепции чистого кода, представленные в данной книге, применимы в разных парадигмах программирования. Многие из этих идей уходят корнями в структурное и функциональное программирование, тогда как другие пришли из объектно-ориентированного мира. Эти концепции помогут вам писать более элегантный и эффективный код в рамках любой парадигмы.

В большинстве рецептов я буду использовать объектно-ориентированные языки и создам симулятор (под названием *MAPPER*), используя объекты в качестве образов для сущностей реального мира. Я буду часто обращаться к этому симулятору на протяжении всей книги. Многие рецепты мотивируют вас использовать поведенческий и декларативный код (см. главу 6 «Декларативный код») вместо кода реализации.

---

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Паттерны объектно-ориентированного проектирования». — СПб., издательство «Питер».

[2] Фарли Д. «Современная программная инженерия. ПО в эпоху эджайла и непрерывного развертывания». — СПб., издательство «Питер».

1.11. Объекты и классы

В большинстве рецептов этой книги речь идет об объектах, а не о классах (хотя созданию классов посвящена целая глава — глава 19 «Иерархии»). Например, рецепт 3.2 называется «Определение сути объектов», а не «Определение сути класса». В книге для отображения реалий окружающего мира служат именно объекты.

Способ создания этих объектов не первостепенен; вы можете использовать классы, прототипы, фабрики, клонирование и т. д. В главе 2 обсуждаются важность сопоставления объектов и необходимость моделирования вещей, которые вы наблюдаете в реальном мире. Для создания объектов во многих языках используются *классы*, которые являются артефактами и неочевидны для реального мира. Они нужны, если вы используете язык классов. Но рецепты данной книги основаны не на них.

1.12. Изменяемость

Чистый код — это не только гарантия правильной работы программ, но и простота его поддержки и будущего улучшения. Опять же, как утверждает Дэйв Фарли в упомянутой выше книге, необходимо быть экспертами в изучении и создании программ, готовых к изменениям. Это серьезный вызов для индустрии высоких технологий, и я надеюсь, что эта книга поможет вам принять его.

ГЛАВА 2

Формулирование аксиом

2.0. Введение

Вот общее определение программного обеспечения (ПО) (https://oreil.ly/MqGxG):

> Инструкции, выполняемые компьютером, в противоположность физическому устройству, на котором они выполняются («аппаратное обеспечение»).

Программное обеспечение определяется от противного; то есть это все, что не является аппаратным обеспечением. Это не совсем удачное определение. Вот еще одно (https://oreil.ly/SVbXv):

> Программное обеспечение — это инструкции, которые указывают компьютеру, что делать. ПО включает в себя весь набор программ, процедур и методов, связанных с работой компьютерной системы. Этот термин был введен в противовес аппаратному обеспечению, то есть физическим компонентам компьютерной системы. Набор инструкций, который указывает аппаратным средствам компьютера выполнять определенную задачу, называется программой или программным обеспечением.

Много десятилетий назад разработчики осознали, что программы — это гораздо больше, чем инструкции. На протяжении всей книги вы будете анализировать поведение системы и придете к пониманию того, что основная цель программ заключается в следующем:

> Имитировать происходящее в возможной реальности.

Эта идея возвращает нас к истокам современных языков программирования, таких как *Simula*.

Simula

Simula был первым объектно-ориентированным языком программирования, который предполагал создание классов (классификацию). Его название явно указывало на цель разработки — создание симулятора. Она по-прежнему справедлива для большинства современных программных продуктов.

В науке вы создаете симуляторы, чтобы понять прошлое и предсказать будущее. Со времен Платона люди пытались построить достоверные модели

реальности. Программное обеспечение можно определить как конструкцию симулятора *MAPPER*:

> Модель: абстрактное частичное и программируемое объяснение реальности (Model: Abstract Partial and Programmable Explaining Reality)

Эта аббревиатура будет часто встречаться в книге. Давайте разберемся, что она собой представляет.

2.1. Почему модель?

Модель — это результат рассмотрения определенного параметра реальности через заданную призму, с заданной точки зрения и с использованием заданной парадигмы. Это не истина в последней инстанции, а скорее наиболее точное имеющееся понимание, основанное на текущих знаниях. Целью программной модели, как и любой другой, является прогнозирование поведения в реальном мире.

Модель

Модель объясняет предмет, который она описывает, используя интуитивные понятия или метафоры. Конечная цель модели — понимание того, как что-то работает. По мнению Питера Наура (Peter Naur) (https://oreil.ly/6FiD8) «программировать — значит разрабатывать теории и модели».

2.2. Почему абстрактное?

Модель возникает из суммы частей. И ее нельзя полностью понять, глядя на отдельные компоненты. Модель основана на соглашениях и описании поведения, а в них не обязательно подробно прописана необходимая реализация.

2.3. Почему программируемое?

Модель необходимо запустить в симуляторе, воспроизводящем желаемые условия. Это может быть как модель Тьюринга (например, современные коммерческие компьютеры), так и квантовый компьютер (компьютеры будущего) или любой другой тип симулятора, способный идти в ногу с эволюцией модели. Вы можете запрограммировать модель так, чтобы она реагировала на ваши действия определенным образом, а затем наблюдать, как она развивается сама по себе.

Модель Тьюринга

Компьютер, основанный на *модели Тьюринга*, представляет собой теоретическую машину, способную выполнять любую вычислимую задачу, для которой можно написать набор инструкций или алгоритм. Машина Тьюринга считается теоретической основой современных компьютерных систем и служит моделью для проектирования и анализа реальных компьютеров и языков программирования.

2.4. Почему частичное?

Для моделирования интересующей вас проблемы вы будете учитывать только некоторые параметры реальности. В научных моделях принято упрощать то, что не имеет отношения к делу, чтобы отделить проблему. При проведении научных экспериментов для проверки гипотез необходимо отделить определенные переменные и зафиксировать остальные.

В симуляторе вы не сможете смоделировать реальность полностью, только соответствующую ее часть. Вам не нужно моделировать весь объект наблюдения (то есть реальный мир), достаточно смоделировать интересующее вас поведение. Многие рецепты в этой книге посвящены проблеме овердизайна, когда в модель включается избыточное количество ненужных деталей.

2.5. Почему объяснение?

Модель должна быть достаточно декларативной, чтобы наблюдать за ее развитием, а также помогать в рассуждениях о моделируемой реальности и предсказании ее поведения. Она должна быть способна объяснить, что она делает и как себя ведет. Многие современные алгоритмы машинного обучения не поясняют, как они получают свои выходные значения (иногда даже больше похожие на галлюцинации), но модели должны объяснять свои действия, пусть даже и не раскрывая конкретных шагов.

Что значит «объяснять»?

Аристотель говорил, что «объяснить — значит найти причины». По его мнению, каждое явление или событие имеет причину или ряд причин, которые его порождают или определяют. Цель науки — выявить и понять причины природных процессов и, исходя из этого, предсказать, как эти процессы поведут себя в дальнейшем.

Для Аристотеля «объяснить» означало выявить и понять эти причины и то, как они взаимодействуют друг с другом, порождая то или иное явление. С другой стороны, термин «предсказать» относится к способности использовать знание причин для того, чтобы предсказывать дальнейшее поведение этого явления.

2.6. Почему мы говорим о реальности?

Модель должна воспроизводить условия, которые происходят в наблюдаемой среде. Конечная цель — предсказать реальные события, как и в любой другой симуляции. В этой книге мы будем часто говорить о реальности, реальном мире и реальных сущностях. Реальный мир станет для вас единственным источником истины.

2.7. Переходим к правилам

Теперь, когда вы получили первое представление о сути программ, вы можете переходить к хорошим практикам моделирования и проектирования. Принципы MAPPER будут встречаться во всех рецептах книги.

В последующих главах вы продолжите открывать для себя принципы, эвристики, рецепты и правила создания совершенных программных моделей, начав с простой введенной здесь аксиомы «Модель: абстрактное частичное и программируемое объяснение реальности». А рабочим определением программного обеспечения в этой книге мы выбрали следующее — это «симулятор, соответствующий условиям MAPPER».

Аксиома

Аксиома — это утверждение или предположение, истинность которого принимается без доказательств. Она позволяет построить логическую основу для рассуждений и выводов, устанавливая набор фундаментальных понятий и взаимозависимостей, которые могут быть использованы для открытия последующих истин.

2.8. Единственный принцип проектирования программ

Основав всю концепцию проектирования программного продукта на одном-единственном правиле, вы сможете сохранить ее простоту и создавать качественные модели. Минимализм и аксиоматичность означают, что весь набор правил можно вывести из одного определения:

> Поведение каждого элемента является частью архитектуры ровно настолько, насколько оно способно помочь в рассуждениях о системе. Поведение элементов отражает то, как они взаимодействуют друг с другом и с окружающей средой. Это, очевидно, является частью нашего определения архитектуры и будет влиять на свойства, присущие системе, такие как ее производительность во время выполнения.
>
> *Басс (Bass) и др. «Software Architecture in Practice»,* (https://learning.oreilly.com/library/view/software-architecture-in/9780136885979/), *2-е издание*[1]

Одним из самых недооцененных признаков качества программного продукта является предсказуемость. В книгах часто пишут, что программы должны быть

[1] Басс Л., Клементс П., Кацман Р. «Архитектура программного обеспечения на практике». — СПб., издательство «Питер».

быстрыми, надежными, устойчивыми, наглядными, безопасными и т. д. При этом предсказуемость редко входит в пятерку приоритетов при проектировании. Попробуйте представить себе процесс проектирования объектно-ориентированного ПО, следующий только одному принципу (как на рис. 2.1): «Каждый объект предметной области должен быть представлен одним объектом в вычислимой модели, и наоборот». Затем попробуйте вывести все правила проектирования, эвристики и рецепты из этой единственной предпосылки, чтобы ваш продукт получился предсказуемым, согласно рецептам этой книги.

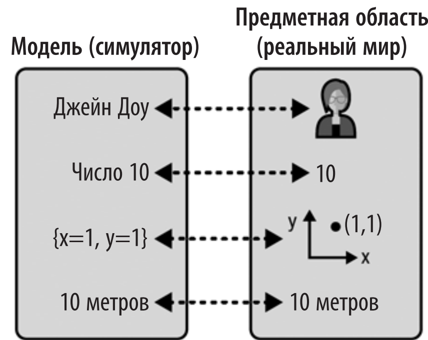

Рис. 2.1. Отношение между объектами модели и сущностями реального мира — 1 к 1

Проблема

Познакомившись с рецептами чистого кода, вы поймете, что большинство используемых в отрасли языковых реализаций игнорируют правило единственной аксиомы, что приводит к колоссальным проблемам. Многие современные языки предназначены для решения трудностей реализации, возникших при создании программ еще 3–4 десятка лет назад, когда ресурсы были скудны и разработчику приходилось вручную оптимизировать вычисления. Сейчас подобное встречается в очень немногих предметных областях. Рецепты из этой книги помогут вам распознать, понять и решить эти проблемы.

Модели в помощь

При построении моделей любого типа важно имитировать реальные условия. Вы можете отслеживать каждый интересующий вас элемент в симуляции и активировать его, наблюдая, меняется ли он в соответствии с реальным поведением. Метеорологи используют математические модели для предсказания и прогнозирования погоды, многие научные дисциплины также опираются на моделирование. Физика ищет универсальные модели, чтобы понимать и предсказывать законы реального мира. С возникновением машинного обучения появилась возможность создавать нечеткие модели для визуализации поведения в реальной жизни.