1 Управление сложностью

В ЭТОЙ ГЛАВЕ

- ✓ Почему системы со временем усложняются.
- ✓ Проблемы объектно-ориентированного проектирования.
- ✓ Почему нужно постоянно совершенствовать архитектуру решения.

В 2010 году я работал в одной замечательной интернет-компании в команде, отвечающей за биллинг (выставление счетов). Основатель компании написал первую версию системы за 10 или 15 лет до моего прихода. Вся логика заключалась в сложных хранимых процедурах SQL Server, каждая из которых состояла из тысяч строк кода. Пришло время выполнить рефакторинг существующей биллинговой инфраструктуры, создав нечто новое, и я не могу сосчитать, сколько часов наш отдел общался с сотрудниками

финансовой команды, чтобы создать архитектуру, которая бы соответствовала их текущим и будущим потребностям.

Отличная новость — мы справились. Благодаря новой версии мы могли добавлять новые продукты или финансовые правила за считаные часы. Финансовая команда была очень довольна нами. Запросы на добавление новых функций, на которые раньше уходили недели, теперь занимали пару дней. Существенно улучшилось и качество. Наш код хорошо поддавался тестированию, поэтому мы редко сталкивались с регрессионными ошибками. Даже наш джуниор мог легко ориентироваться в коде и чувствовать себя достаточно уверенно, чтобы вносить важные изменения. Одним словом, наш новый проект был *простым*.

Я занимаюсь разработкой объектно-ориентированных программных систем уже 20 лет и понял, что в объектно-ориентированной системе без надлежащего проектирования даже простые вещи оказываются слишком сложными. Но так не должно быть.

1.1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ИСПЫТАНИЕ ВРЕМЕНЕМ

Объектно-ориентированное программирование — отличный выбор при реализации сложных программных продуктов, где обязательными условиями являются гибкость и удобство сопровождения. Однако просто выбрать объектно-ориентированный язык для своего проекта недостаточно. Необходимо правильно его использовать.

К счастью, нам не нужно изобретать оптимальные приемы работы с объектно-ориентированными системами с нуля, поскольку профессиональное сообщество уже обладает обширными знаниями. Если вы мало что знаете о существующих приемах или хотите освежить их в памяти, то эта книга идеально вам подойдет, и вам стоит прочитать ее от начала до конца, в том числе примеры кода. Если вы уже более опытный инженер и знаете о существующих приемах, то эта книга поможет вам взглянуть на них по-другому, более прагматично и, надеюсь, натолкнет вас на интересные мысли.

Практически у любого разработчика при создании информационной системы на объектно-ориентированном языке возникает несколько общих вопросов.

- Достаточно ли проста эта реализация или мне следует предложить еще более элегантную абстракцию?
- Этот класс проходит через множество состояний в течение своего жизненного цикла. Как мне гарантировать его консистентность?
- Как моделировать взаимодействие моей системы и внешнего веб-приложения?
- Допустимо ли связать классы или это плохая идея?

Эта книга называется «Простое объектно-ориентированное проектирование», поскольку *простые объектно-ориентированные проекты всегда легче сопровождать.* Мало разработать простой проект — важно сохранить его в таком виде. За эти годы я многому научился на примере своих удачных и неудачных решений, и в этой книге поделюсь набором паттернов, которые помогают мне создавать объектно-ориентированные информационные системы, которые легко сопровождать и развивать.

1.2. ПРОЕКТИРОВАНИЕ ПРОСТЫХ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ СИСТЕМ

> «По мере развития информационных систем их сложность возрастает, если не предпринимать усилий по ее удержанию или снижению».

Эту мысль высказал Мэнни Леман (Manny Lehman) в своей работе 1984 года *On understanding laws, evolution, and conservation in the large-program life cycle* (https://dl.acm.org/doi/10.1016/0164-1212%2879%2990022-0). Доработка систем любого типа — дело непростое. Мы знаем, что код имеет тенденцию засоряться со временем и требует усилий по сопровождению. И несмотря

на 40 лет прогресса, сопровождаемость программных продуктов остается сложной задачей.

По сути, степень сопровождаемости — это количество усилий, которые вы прикладываете для выполнения таких задач, как изменение бизнес-правил, добавление функций, выявление ошибок и исправление системы. Программное обеспечение с высокой степенью сопровождаемости позволяет разработчикам выполнять эти задачи, прикладывая разумные усилия, в то время как низкая сопровождаемость все усложняет, отнимает много времени и плодит ошибки.

На удобство сопровождения влияет множество факторов: от чрезмерно сложного кода до управления зависимостями, плохо продуманных абстракций и некачественной модульности. Системы со временем естественным образом усложняются, поэтому постоянное расширение кода без учета последствий для сопровождения может быстро запутать базу.

Последовательная борьба с увеличением сложности очень важна, даже если кажется трудоемкой. И я знаю, что она требует больше усилий, чем просто создание дампа (dumping code). Но поверьте, разработчики чувствуют себя гораздо хуже, когда целый день разбираются с полотнищами спагетти-кода. Наверняка вам приходилось работать с кодовыми базами, которые было сложно сопровождать. Мне — да. Любые действия в таких системах занимают много времени. Вы не можете найти место, в котором надо писать свою часть; весь код, который вы пишете, кажется временным решением; вы не можете написать автоматизированные тесты, поскольку код не поддается тестированию; вы всегда боитесь, что тот или иной элемент не будет работать должным образом, поскольку не уверены в том, какие изменения вообще можно вносить, и т. д.

Что же такое простой объектно-ориентированный проект? Исходя из моего опыта, это проект, который обладает следующими шестью характеристиками (рис. 1.1):

- простой код;
- консистентные объекты;
- качественное управление зависимостями;
- хорошие абстракции;
- правильная работа с внешними зависимостями и инфраструктурой;
- продуманная модульность.

Эти идеи могут показаться вам знакомыми. Все они популярны в объектно-ориентированных системах. Теперь вкратце рассмотрим, что я имею в виду под каждой из них и что происходит, когда мы теряем контроль.

Рис. 1.1. Характеристики простого объектно-ориентированного проекта

1.2.1. Простой код

Реализация простых методов и классов — отличный способ начать заниматься объектно-ориентированным проектированием. Рассмотрим метод, который начинался с малого количества строк и условных операторов, но со временем разросся и теперь содержит сотни строк и одни операторы `if` внутри других. Сопровождать такой метод очень сложно.

Интересно, что классы и методы поначалу просты и управляемы. Но если мы не стараемся поддерживать их в таком состоянии, то они становятся сложными для понимания и сопровождения (рис. 1.2). В запутанных реализациях кода сложно разбираться, что неизбежно породит ошибки. Вдобавок сопровождение, рефакторинг и тестирование такого кода требуют больших усилий, поскольку разработчики боятся что-либо сломать и пытаются определить все возможные тестовые сценарии.

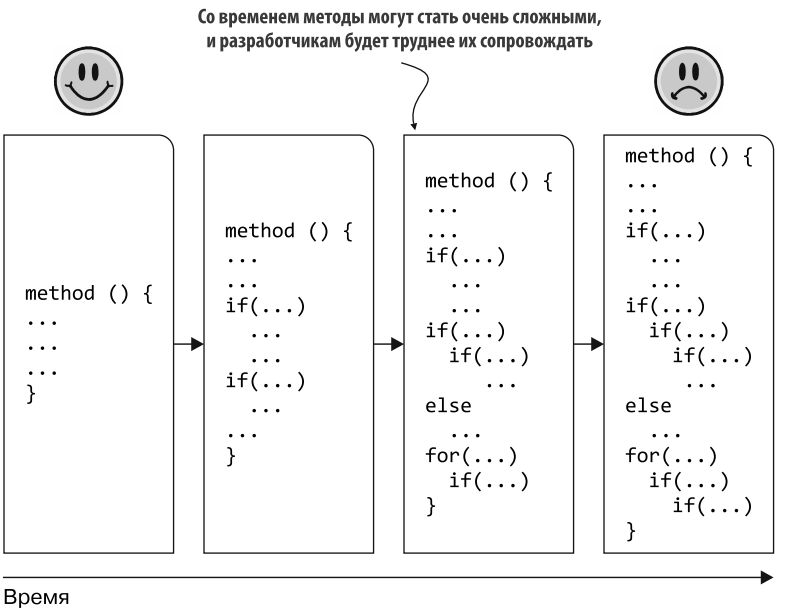

Рис. 1.2. Простой код со временем усложняется, и, как следствие, его очень трудно сопровождать

Существует множество способов уменьшить сложность класса или метода. Например, ясные и выразительные имена переменных помогают разработчикам лучше понять их назначение. Однако в этой части книги я утверждаю, что правило № 1 для сохранения простоты классов и методов звучит так: они должны быть *небольшими*. Метод не должен быть слишком длинным. В классе не должно быть слишком много методов. Мелкие единицы кода всегда легче сопровождать и развивать.

1.2.2. Согласованные объекты

Гораздо проще работать над системой, когда вы можете быть уверены, что объекты консистентны и любая попытка рассогласовать их будет пресечена. Когда согласованность не учитывается при проектировании, объекты могут находиться в недопустимых состояниях, что приводит к ошибкам и проблемам с сопровождением.

Рассмотрим класс `Basket` в системе электронной торговли, отслеживающий товары, которые покупает человек, и их конечную стоимость. Итоговая стоимость должна обновляться каждый раз, когда мы добавляем товар в корзину или удаляем его из нее. Кроме того, корзина должна отклонять некорректные запросы клиентов, например добавление одного товара несколько раз или удаление товара, которого нет в корзине.

На рис. 1.3 слева показана защищенная корзина: товары можно добавлять или удалять только обращением к корзине. Она полностью контролирует содержимое и обеспечивает его согласованность. Справа — незащищенная корзина, открывающая неограниченный доступ к ее содержимому. Из-за отсутствия контроля эта корзина не всегда может обеспечить согласованность.

Таким образом, хорошая архитектура гарантирует, что объекты никогда не будут находиться в неконсистентном состоянии. Согласованность может быть нарушена вследствие многих действий, например при использовании неправильных сеттеров, которые обходят проверки согласованности, или в случае отсутствия гибких механизмов валидации, подробнее о которых мы поговорим позже.

Рис. 1.3. Две корзины: одна контролирует действия, которые в ней происходят, а другая — нет. Управление состоянием и согласованностью — основополагающий фактор

1.2.3. Качественное управление зависимостями

В крупномасштабных объектно-ориентированных системах управление зависимостями становится критически важным элементом удобства сопровождения. Когда в системе с высокой степенью связанности никого не интересует, как классы связаны между собой, любое простейшее изменение может привести к непредсказуемым последствиям.

На рис. 1.4 показано, как на класс `Basket` могут повлиять изменения в любом из зависящих от него классов: `DiscountRules`, `Product` и `Customer`. Даже изменение в классе `DiscountRepository`, транзитивной зависимости, может повлиять на `Basket`. Например, если класс `Product` часто меняется, то `Basket` всегда рискует тоже измениться.

Простые объектно-ориентированные проекты направлены на минимизацию зависимостей между классами. Чем меньше они зависят друг от друга и чем меньше знают друг о друге, тем лучше. Кроме того, правильное управление зависимостями гарантирует, что классы будут максимально зависеть от стабильных

компонентов, вероятность изменения которых невелика, как и вероятность спровоцированных ими каскадных изменений.

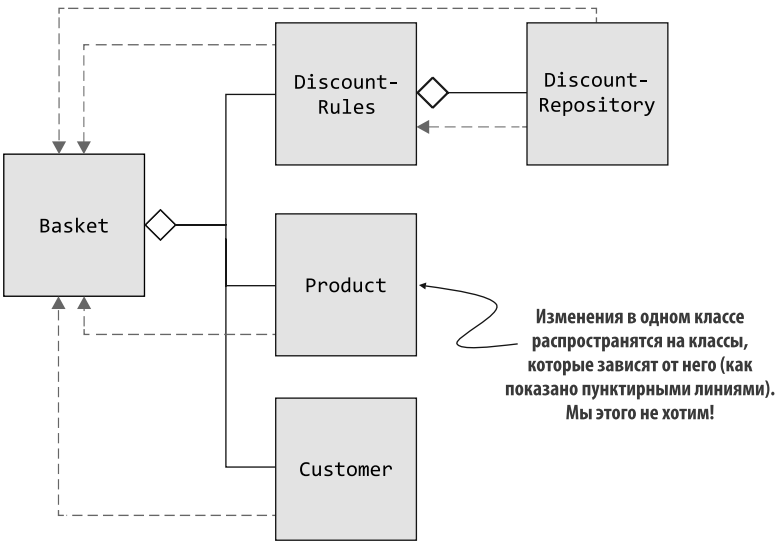

Рис. 1.4. Управление зависимостями и распространение изменений

1.2.4. Хорошие абстракции

Простой код всегда предпочтительнее, но расширяемость может потребовать большего. Расширение класса путем добавления большего количества кода в какой-то момент перестает быть эффективным и превращается в обузу.

Представьте, что в одном классе или методе реализовано 30 или 40 различных бизнес-правил. Я показываю это на рис. 1.5. Обратите внимание, как класс `DiscountRules`, отвечающий за применение различных скидок в нашей системе электронной торговли, увеличивается по мере появления новых правил скидок, что значительно усложняет его сопровождение. Хорошее архитектурное решение позволяет разработчикам пользоваться абстракциями, которые помогают им развивать систему, не усложняя существующие классы.

Рис. 1.5. Класс, не имеющий абстракций, бесконечно усложняется

1.2.5. Правильная работа с внешними зависимостями и инфраструктурой

Простые объектно-ориентированные проекты отделяют доменный слой, содержащий бизнес-логику, от кода, необходимого для взаимодействия с внешними зависимостями. На рис. 1.6 слева показаны классы домена, а справа — классы, обеспечивающие взаимодействие со сторонними системами и инфраструктурой.

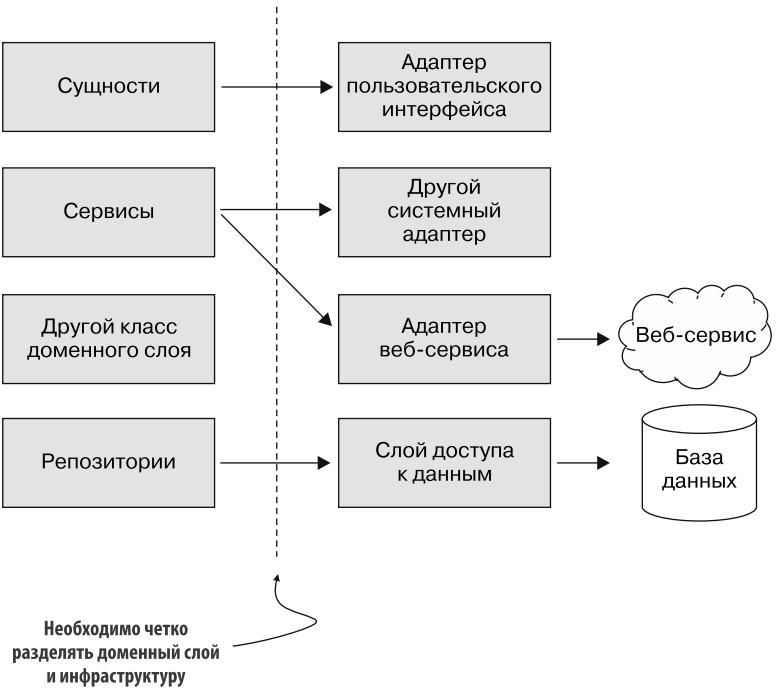

Рис. 1.6. Архитектура информационной системы, в которой инфраструктура отделена от доменного слоя или бизнес-логики

Если детали инфраструктуры проникнут в ваш доменный слой, это может помешать вам вносить изменения в инфраструктуру. Представьте, что весь код для доступа к базе данных разбросан по кодовой базе. Теперь вам нужно добавить слой кэширования,

чтобы ускорить время отклика приложения. Возможно, для этого вам придется изменить код повсеместно.

Проблема заключается в том, чтобы абстрагироваться от несущественных или внешних аспектов вашей инфраструктуры и при этом использовать предоставляемые ею ценные возможности. Например, если вы используете реляционную базу данных, такую как Postgres, то можете захотеть скрыть ее присутствие от кода предметной области, но при этом иметь возможность применять ее уникальные функции, которые повышают производительность или быстродействие.

Почему я называю это инфраструктурой?

Я использую термин *«инфраструктура»* для обозначения любой зависимости от внешних систем и ресурсов, таких как вебсервисы, базы данных, сторонние API и все, что находится за пределами вашей системы. При возникновении такой зависимости необходимо написать код, соединяющий вашу систему с внешней системой или ресурсом. В главе 6 мы сосредоточимся на гибком написании такого «связующего кода», чтобы он не навредил остальной части вашего проекта.

1.2.6. Продуманная модульность

Информационные системы развиваются, и уместить все в одном компоненте или модуле сложно. В простых объектно-ориентированных проектах большие системы разделяются на независимые компоненты, которые взаимодействуют для достижения общей цели.

Благодаря разделению систем на более мелкие компоненты их легче сопровождать и понимать. Вдобавок это помогает разным командам работать над отдельными компонентами без конфликтов. Управлять мелкими компонентами и тестировать их — тоже более простая задача.

Рассмотрим программный продукт с тремя предметными областями: «Счет» (Invoice), «Выставление счетов» (Billing) и «Доставка»

(Delivery). Они должны работать вместе, причем «Счет» и «Доставка» требуют информации от области «Выставление счетов».

На рис. 1.7 слева показана система без модулей, в которой свободно перемешиваются классы из разных доменных областей. По мере увеличения сложности такая система становится неуправляемой. В правой части рисунка показана та же система, разделенная на модули: «Выставление счетов», «Счет» и «Доставка». Модули взаимодействуют через интерфейсы, что позволяет клиентам использовать только необходимые элементы, не разбираясь во всей доменной области.

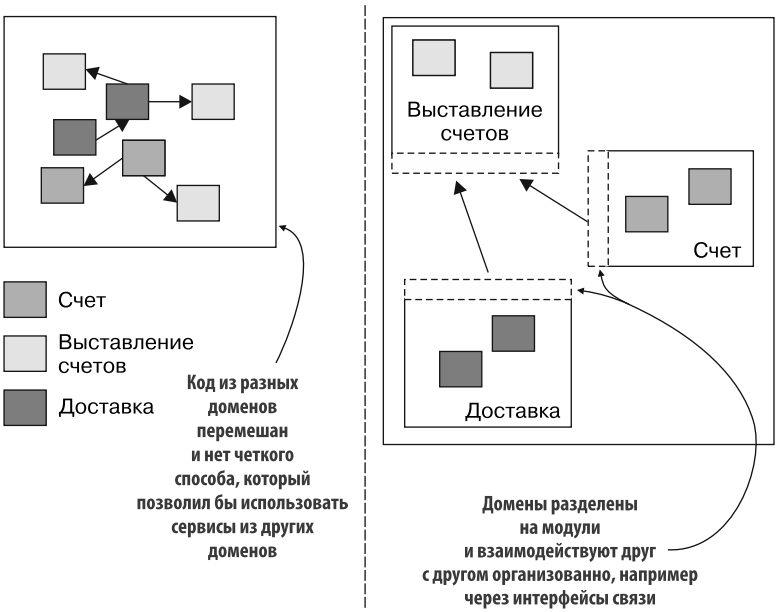

Рис. 1.7. Две системы с различными подходами к модульному построению

Определить нужный уровень детализации для модуля или то, как должен выглядеть его публичный интерфейс, — непростая задача. Мы поговорим об этом позже.