5 Проектирование бизнес-логики в микросервисной архитектуре

В этой главе

- Применение шаблонов организации бизнес-логики, таких как сценарий транзакции и доменная модель.
- Проектирование бизнес-логики с помощью шаблона «Агрегат» (предметно-ориентированное проектирование).
- Применение шаблона «Доменное событие» в микросервисной архитектуре.

Сердцем промышленных приложений является бизнес-логика, которая реализует бизнес-правила. Разработка сложной бизнес-логики всегда сопряжена с определенными трудностями. Приложение FTGO реализует довольно замысловатую бизнес-логику, особенно для управления заказами и доставкой. Мэри поощряла свою команду применять принципы объектно-ориентированного проектирования, поскольку, исходя из ее опыта, это лучший способ реализации сложной бизнес-логики. На некоторых участках приложения использовался процедурный шаблон «Сценарий транзакции». Но бóльшая часть кода была реализована в соответствии с объектно-ориентированной доменной моделью, которая накладывалась на базу данных с помощью JPA.

В микросервисной архитектуре разрабатывать сложную бизнес-логику оказывается еще труднее, потому что она распределена между разными микросервисами. Вам необходимо решить две ключевые проблемы. Типичная доменная модель выглядит как паутина из связанных между собой классов. В монолитных приложениях в этом нет ничего плохого, но в микросервисной архитектуре, где классы разбросаны по разным сервисам, нужно избавиться от ссылок на объекты, которые пересекают

границы сервисов. Еще одна проблема заключается в проектировании бизнес-логики, которая работает в рамках ограничений, накладываемых работой с транзакциями в микросервисной архитектуре. Вы можете применять ACID-транзакции внутри одного сервиса, но, как говорилось в главе 4, для обеспечения согласованности данных между сервисами следует использовать шаблон «Повествование».

К счастью, для преодоления этих трудностей можно воспользоваться шаблоном «Агрегат» из состава DDD. Он структурирует бизнес-логику приложения в виде набора агрегатов. *Агрегат* — это кластер объектов, с которыми можно обращаться как с единым целым. Есть две причины, почему агрегаты могут пригодиться при разработке бизнес-логики в микросервисной архитектуре.

- Агрегаты исключают любую возможность того, что ссылки на объекты могут выйти за рамки одного сервиса, потому что межагрегатная ссылка — это скорее значение первичного ключа, а не объектная ссылка.
- Транзакция может создать или обновить лишь один агрегат, поэтому агрегаты соответствуют ограничениям транзакционной модели микросервисов.

Благодаря этому ACID-транзакция никогда не выйдет за пределы одного сервиса.

Я начну эту главу с описания разных способов организации бизнес-логики — шаблонов «Сценарий транзакции» и «Доменная модель». Затем вы познакомитесь с концепцией агрегатов из DDD и узнаете, почему они являются хорошими строительными блоками для бизнес-логики сервисов. После этого я опишу шаблон «Доменная модель» и объясню, почему сервису следует публиковать свои события. В конце главы будут представлены несколько примеров бизнес-логики из сервисов `Kitchen` и `Order`.

Рассмотрим шаблоны организации бизнес-логики.

5.1. Шаблоны организации бизнес-логики

На рис. 5.1 показана архитектура типичного сервиса. Как говорилось в главе 2, бизнес-логика является ядром шестигранной архитектуры. Ее окружают входящие и исходящие адаптеры. *Входящий адаптер* обрабатывает запросы от клиентов и вызывает бизнес-логику. *Исходящий адаптер*, который сам вызывается бизнес-логикой, обращается к другим сервисам и приложениям.

Этот сервис состоит из бизнес-логики и следующих адаптеров:

- *адаптера REST API* — входящего адаптера, который реализует REST API для вызова бизнес-логики;
- `OrderCommandHandlers` — входящего адаптера, который потребляет из канала командные сообщения и вызывает бизнес-логику;
- *адаптера базы данных* — исходящего адаптера, который вызывается бизнес-логикой для доступа к базе данных;
- *адаптера публикации доменных событий* — исходящего адаптера, который публикует события для брокера сообщений.

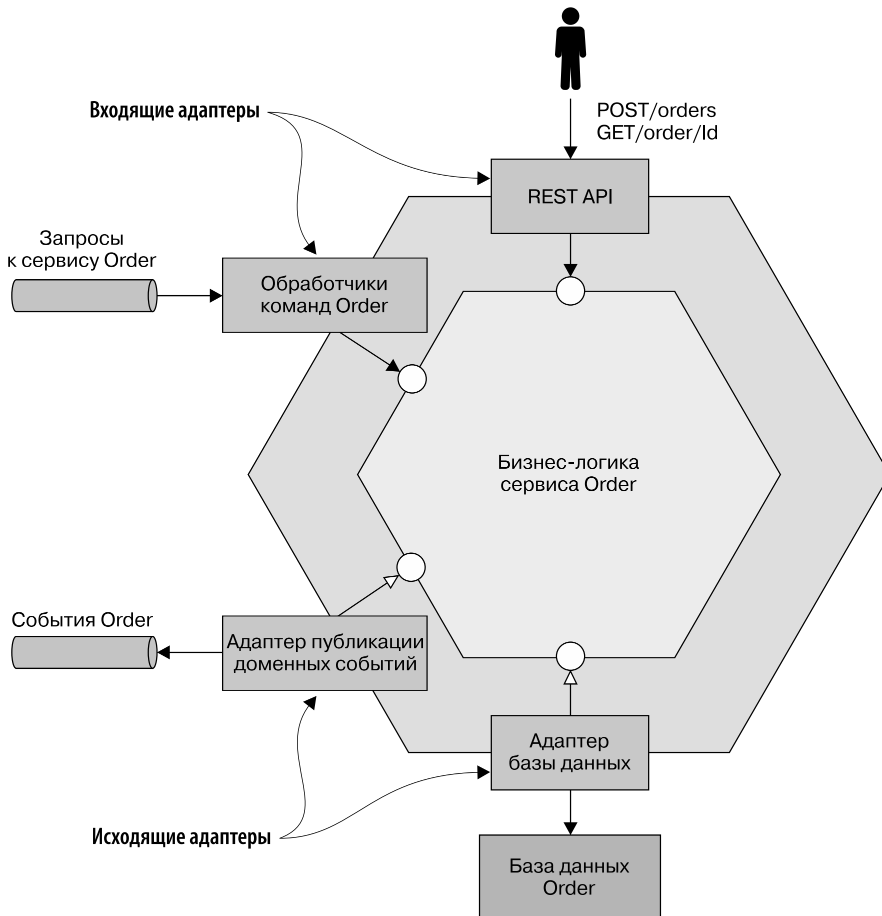

Рис. 5.1. Сервис Order имеет шестигранную архитектуру. Он состоит из бизнес-логики и одного или нескольких адаптеров для доступа к внешним приложениям или другим сервисам

Бизнес-логика обычно оказывается самой сложной частью сервиса. Вы должны осознанно организовать ее таким образом, который лучше всего подходит для вашего приложения. Я уверен, что вам уже знакомо разочарование от поддержки плохо структурированного кода, написанного кем-то другим. Большинство приложений уровня предприятия написаны на объектно-ориентированных языках, таких как Java, поэтому они состоят из классов и методов. Но использование объектно-ориентированного языка вовсе не означает, что бизнес-логика имеет объектно-ориентированную структуру. Ключевое решение, которое вам придется принять в ходе разработки, заключается в том, какой подход лучше применять — объектно-ориентированный или процедурный. Существует два основных шаблона для организации бизнес-логики: процедурный «Сценарий транзакции» и объектно-ориентированный, который называется «Доменная модель».

5.1.1. Проектирование бизнес-логики с помощью шаблона «Сценарий транзакции»

Я большой сторонник объектно-ориентированного подхода, но в некоторых ситуациях он излишен — например, при разработке простой бизнес-логики. В таких случаях лучше писать процедурный код, используя шаблон, который Мартин Фаулер в своей книге *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002)[1] называет сценарием транзакции. Вместо объектно-ориентированного проектирования вы создаете метод под названием *«Сценарий транзакции»*, который обрабатывает запросы из уровня представления. Важная характеристика этого подхода — то, что классы, реализующие поведение, отделены от классов, которые хранят состояние (рис. 5.2).

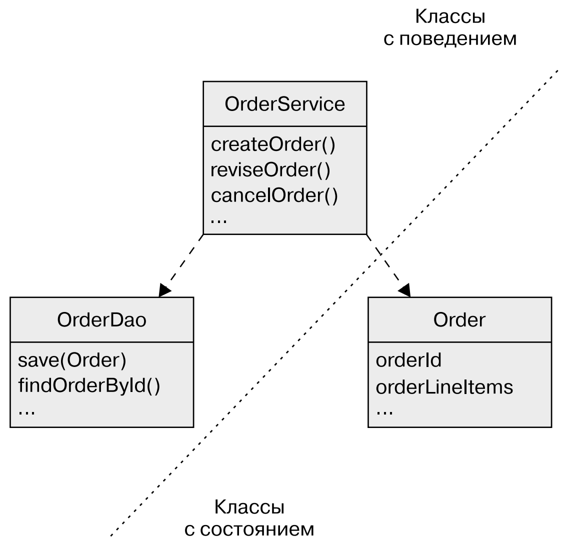

Рис. 5.2. Организация бизнес-логики в виде сценариев транзакции. В такой архитектуре один набор классов обычно реализует поведение, а другой хранит состояние. Сценарии транзакции организованы в виде классов, у которых нет состояния. Они применяют классы данных, которые, как правило, не обладают поведением

При использовании этого шаблона сценарии обычно размещаются в классе сервиса (в данном случае `OrderService`). Класс сервиса имеет по одному методу для каждого запроса или системной операции. Метод реализует бизнес-логику для определенного запроса. Он обращается к БД с помощью объектов доступа к данным (data access object, DAO), таких как `OrderDao`. Здесь примером такого объекта является класс `Order`, он предназначен исключительно для работы с данными, и у него почти (или совсем) нет никакого поведения.

[1] *Фаулер М.* Шаблоны корпоративных приложений. — М.: Вильямс, 2016.

Шаблон «Сценарий транзакции»

Организует бизнес-логику в виде набора процедурных сценариев транзакций, по одному для каждого типа запросов.

Этот стиль проектирования в основном является процедурным, но при этом использует несколько возможностей объектно-ориентированных языков программирования. Вы бы с помощью этого подхода писали программы на C и других языках без поддержки объектно-ориентированного программирования (ООП). Тем не менее в процедурном проектировании, если оно применяется в подходящей ситуации, нет ничего постыдного. Оно хорошо подходит для простой бизнес-логики, но сложную логику с его помощью лучше не реализовывать.

5.1.2. Проектирование бизнес-логики с помощью шаблона «Доменная модель»

Простота процедурного подхода может показаться довольно соблазнительной. Вы можете писать код без тщательного продумывания организации классов. Но проблема в том, что при довольно значительном усложнении бизнес-логики поддержка вашего кода превратится в сплошной кошмар. Как и монолитные приложения, сценарии транзакций склонны постоянно разрастаться. Поэтому, если ваше приложение не является предельно простым, следует воздерживаться от написания процедурного кода. Вместо этого стоит применять доменную модель и вести разработку в объектно-ориентированном стиле.

Шаблон «Доменная модель»

Организует бизнес-логику в виде объектной модели, состоящей из классов с состоянием и поведением.

В объектно-ориентированном проектировании бизнес-логика состоит из объектной модели — сети относительно небольших классов. Эти классы обычно напрямую соотносятся с концепциями из проблемной области. В такой архитектуре некоторые классы обладают либо состоянием, либо поведением, но многие имеют и то и другое, что является признаком хорошо спроектированного класса. Пример шаблона «Доменная модель» показан на рис. 5.3.

Как и в случае с шаблоном «Сценарий транзакции», у класса `OrderService` предусмотрено по одному методу для каждого запроса или системной операции. Но при использовании доменной модели методы сервисов обычно получаются более простыми. Это связано с тем, что существенная часть бизнес-логики делегируется доменным объектам. Метод сервиса может, например, извлечь доменный объект из базы данных и вызвать один из его собственных методов. В этом примере класс

Order обладает как состоянием, так и поведением. Более того, его состояние является приватным, и доступ к нему осуществляется лишь через его методы.

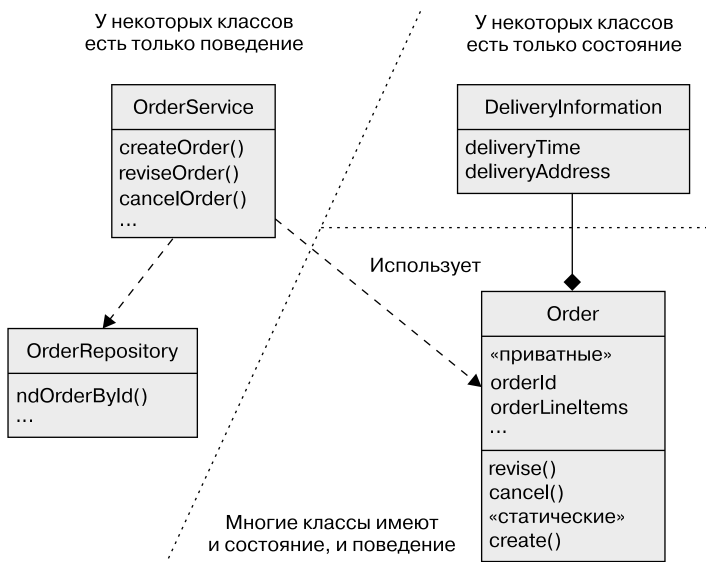

Рис. 5.3. Организация бизнес-логики в виде доменной модели. Большинство классов имеют и состояние, и поведение

Применение объектно-ориентированного проектирования имеет ряд преимуществ. Во-первых, это упрощает понимание и поддержку архитектуры. Вместо одного большого класса, который берет на себя все функции, сервис состоит из нескольких мелких классов, у каждого из которых есть свой небольшой набор обязанностей. Кроме того, такие классы, как Account, BankingTransaction и OverdraftPolicy, довольно точно отражают реальный мир, благодаря чему их роль в архитектуре проще понять. Во-вторых, нашу объектно-ориентированную архитектуру легче тестировать: каждый класс может и должен тестироваться отдельно. И наконец, объектно-ориентированный код проще расширять, поскольку в нем можно использовать хорошо известные шаблоны проектирования, такие как «Стратегия» и «Шаблонный метод», которые позволяют расширять компонент без изменения его кода.

Шаблон «Доменная модель» хорошо себя зарекомендовал, но у него есть целый ряд проблем, особенно в контексте микросервисной архитектуры. Чтобы разобраться с ними, нужно использовать более узкую версию ООП, известную как DDD.

5.1.3. О предметно-ориентированном проектировании

Предметно-ориентированное проектирование (DDD), описанное в книге Эрика Эванса *Domain-Driven Design*, — это более узкая разновидность ООП, предназначенная для разработки сложной бизнес-логики. Мы познакомились с DDD в главе 2 при обсуждении поддоменов и того, насколько они подходят для разбиения при-

ложений на сервисы. В DDD каждый сервис имеет собственную доменную модель, что позволяет избежать проблем с единой доменной моделью, которая охватывает все приложение. Поддомены и связанная с ними концепция изолированного контекста — это два стратегических шаблона DDD.

В DDD есть также тактические шаблоны, которые служат строительными блоками для доменных моделей. Каждый шаблон представляет собой роль, которую класс играет в доменной модели, и описывает характеристики этого класса. Разработчики широко применяют следующие строительные блоки.

- *Сущность* — объект, обладающий устойчивой идентичностью. Две сущности, чьи атрибуты имеют одинаковые значения, — это все равно разные объекты. В приложении Java EE классы, которые сохраняются с помощью аннотации `@Entity` из JPA, обычно представляют собой сущности DDD.
- *Объект значений* — объект, представляющий собой набор значений. Два объекта значений с одинаковыми атрибутами взаимозаменяемы. Примером таких объектов может служить класс `Money`, который состоит из валюты и суммы.
- *Фабрика* — объект или метод, реализующий логику создания объектов, которую ввиду ее сложности не следует размещать прямо в конструкторе. Фабрика также может скрывать конкретные классы, экземпляры которых создает. Она реализуется в виде статического метода или класса.
- *Репозиторий* — объект, предоставляющий доступ к постоянным сущностям и инкапсулирующий механизм доступа к базе данных.
- *Сервис* — объект, реализующий бизнес-логику, которой не место внутри сущности или объекта значений.

Многие разработчики используют эти строительные блоки. Некоторые из них поддерживаются такими фреймворками, как JPA и Spring. Но есть еще одна концепция, которую обычно игнорируют все (и я тоже!), за исключением истинных ценителей DDD. Речь идет об агрегатах. Несмотря на свою непопулярность, этот строительный блок чрезвычайно полезен при разработке микросервисов. Давайте рассмотрим некоторые неочевидные проблемы классического ООП, которые можно решить с помощью агрегатов.

5.2. Проектирование доменной модели с помощью шаблона «Агрегат» из DDD

В традиционном объектно-ориентированном проектировании доменная модель описывает набор классов и отношения между ними. Классы обычно сгруппированы в пакеты. Например, на рис. 5.4 показана часть доменной модели приложения FTGO. Это типичная доменная модель, представляющая собой паутину взаимосвязанных классов.

Этот пример содержит несколько классов, которые соотносятся с бизнес-объектами: `Consumer`, `Order`, `Restaurant` и `Courier`. Но что интересно, в традиционной доменной модели не хватает четких границ между разными бизнес-объектами.