3 Основные концепции предметно-ориентированного проектирования

В этой главе

- Самые важные аспекты предметно-ориентированного проектирования с точки зрения безопасности.
- Модели как строгое упрощение предметной области.
- Объекты-значения, сущности и агрегаты.
- Доменные модели как единый язык.
- Ограниченные контексты и семантические границы.

За годы, на протяжении которых мы занимаемся разработкой программного обеспечения, мы нашли много источников вдохновения, в том числе общих для всех нас. Одним из важнейших стало *предметно-ориентированное проектирование* (Domain-Driven Design, DDD).

DDD устанавливает чуть более высокую планку для разработки большинства систем. Нам встречалось много проектов, где главный руководящий принцип звучал как «просто сделай, чтобы оно работало». Когда находили программную ошибку, решение состояло в добавлении инструкции `if`. Несмотря на то что ошибки редко были порождены исключительно локальными участками кода, в проблемы никто не вникал и система была основана на неполной и несогласованной модели.

> Предметно-ориентированное проектирование — это подход к разработке ПО, в котором мы:
>
> 1) сосредотачиваемся на основной предметной области;
> 2) исследуем модели в творческом сотрудничестве между теми, кто имеет опыт в предметной области, и теми, кто разрабатывает ПО;
> 3) разговариваем на едином языке в рамках четко ограниченного контекста.
>
> *Eric Evans, Domain-Driven Design Reference (Dog Ear Publishing, 2014)*

DDD утверждает, что мы должны стремиться не просто к созданию рабочей системы, а к настоящему пониманию того, что мы на самом деле создаем. Подчеркнем слово *«что»*. DDD делает упор на глубоком понимании предметной области, а не только самого решения. С нашей точки зрения, прелесть принципов DDD в том, что они заставляют нас воплотить это понимание в коде, — в результате код использует тот же язык, что и проблема, которую вы решаете. Мы считаем, что акцент на глубоком понимании помогает нам совершенствоваться как разработчикам. А то, что данный подход серьезно воздействует на безопасность, стало очевидным намного позже.

Эта глава посвящена лишь некоторым аспектам DDD. Это огромная и многосторонняя тема, охват которой — от написания кода до системной интеграции, от анализа требований до тестирования. Она тесно связана с другими гибкими методологиями и процессами. О DDD написано много книг и огромное количество статей, поэтому всеобъемлющее рассмотрение этого подхода в рамках одной главы было бы невозможным. Вместо этого мы сосредоточимся на тех аспектах DDD, которые, как показывает наш опыт, могут улучшить безопасность.

Если вы незнакомы с DDD, то здесь сможете понять этот подход, что пригодится вам в следующих главах. Эта глава также может служить справочником. Позже мы воспользуемся разными аспектами DDD для усиления безопасности, поэтому вы можете обращаться к этому материалу, если нужно будет освежить знания об объектах-значениях, агрегатах, картах контекстов или любой другой концепции DDD.

Мы рекомендуем вам познакомиться с этой темой поближе, так как она охватывает далеко не только безопасность. Хорошим началом будет мини-книга *Domain-Driven Design Quickly*[1]. Для начинающих подойдет также *Patterns, Principles and Practices of Domain-Driven Design* (Wrox, 2015) Скотта Миллетта. Если вы хотите глубоко изучить эту тему, почитайте основополагающую книгу Эрика Эванса «Предметно-ориентированное проектирование (DDD)»[2] — в ней вы найдете исчерпывающий материал.

---

[1] Domain-Driven Design Quickly (InfoQ, 2006) можно загрузить бесплатно в формате PDF по ссылке https://www.infoq.com/minibooks/domain-driven-design-quickly.

[2] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — 448 с.

Если вы уже в какой-то степени знакомы с DDD, воспользуйтесь этой главой для освежения своих знаний. И даже если вы опытный предметно-ориентированный проектировщик, все равно почитайте ее, так как здесь затронуты аспекты, которые мы хотели бы акцентировать, — они будут использоваться в дальнейшем для усиления безопасности. Имейте также в виду, что некоторые идеи будут представлены в несколько сжатом виде и могут показаться немного искаженными. Мы не претендуем на полноту, а лишь стремимся предоставить вам материал, которого будет достаточно для обсуждения безопасности.

Мы рассмотрим *доменные модели*, которые лежат в основе системной разработки в стиле DDD. Доменные модели закладывают недвусмысленный, строгий фундамент описания функций системы. Это представляет определенный интерес с точки зрения безопасности. Описывая обязанности системы, вы заодно получаете отличную возможность определить, чем она не должна заниматься.

При моделировании и реализации моделей в коде полезно иметь под рукой какие-нибудь составные элементы. Доменные модели обычно основаны на объектах-значениях и сущностях. Более крупные структуры, как правило, представлены в виде агрегатов. Использование этих элементов сделает ваш код более строгим и менее склонным к появлению уязвимостей.

Если отойти от отдельной системы и рассмотреть уровень интеграции, то DDD предоставляет такие инструменты, как ограниченные контексты и карты контекстов. С их помощью вам будет легче обеспечить безопасность и тесную интеграцию между несколькими системами. Предметно-ориентированное проектирование основано на строгих доменных моделях, поэтому вначале мы покажем, как их создавать, чтобы с их помощью хорошо понять задачу, которую можно решить посредством вашего ПО.

3.1. Модели как средства обеспечения более глубокого понимания

Сначала объясним, что представляют собой модели, лежащие в основе DDD. В системной разработке слово *«модель»* может означать много разных вещей: диаграммы потоков в UML, способ хранения информации в таблицах базы данных и т. д. В DDD модель описывает основные бизнес-аспекты, с которыми вы имеете дело, в виде определенного набора концепций. Зачем нам нужны такие модели и как они должны выглядеть?

Все мы знаем, что панацеи не существует, и DDD не исключение. Чтобы не вводить никого в заблуждение, всегда необходимо отмечать те случаи, когда методика или методология *не дает* существенных преимуществ, при этом следует указывать обстоятельства, для которых она идеально подходит. Если вы проектируете сетевой маршрутизатор или систему управления багажом, внешние обстоятельства могут сильно разниться. В первом случае DDD мало чем поможет, а во втором сможет принести немалую пользу.

В примере с сетевым маршрутизатором самой важной является техническая проблема — достижение довольно высокой пропускной способности ввода/вывода и довольно низкой латентности, что вовсе не простая задача. Если вам не удастся с ней справиться, у вас получится продукт, который никто не захочет покупать. Сетевая производительность — важнейшая характеристика маршрутизатора. DDD может помочь вам с моделированием очередей пакетов и таблиц маршрутизации, но не с пропускной способностью или латентностью.

Для сравнения рассмотрим пример с системой управления багажом в аэропорту. Ее техническая реализация будет использовать те же базы данных, очереди сообщений и фреймворки графического пользовательского интерфейса, что и большинство других систем. Конечно, все это привносит немалую сложность, но, скорее всего, основная проблема будет в другом. Ваша система должна уметь описывать прохождение багажа из пункта регистрации в самолет по лентам транспортеров и погрузочных машин. Если это описание получится некорректным, багаж может опоздать на нужный рейс или вылететь не в том направлении. Это может раздосадовать пассажиров и нанести компании репутационный и финансовый ущерб. Что еще хуже, на кону важные аспекты безопасности. По очевидным причинам багаж допускается на самолет только в том случае, когда там уже находится его владелец. Если багаж уже зарегистрирован, а пассажир не явился, система должна позаботиться о выгрузке соответствующих чемоданов. Если она разработана с ошибками, существует риск того, что ее можно будет заставить погрузить чемодан на определенный рейс или оставить его на борту, даже если его следует выгрузить, — иногда это может иметь серьезные последствия с точки зрения безопасности.

Если вам не удастся получить глубокое и четкое понимание того, как обрабатывается багаж, вы создадите дефектную систему. Но хуже всего то, что это может быть вредно для компании и потенциально опасно для клиентов. Может дойти до того, что система окажется совершенно бесполезной и аэропорт скорее закроется, чем примет ее в эксплуатацию. Это не гипотетический пример: открытие Денверского аэропорта, которое должно было состояться в 1990-м, отложили на полтора года из-за изъянов в системе управления багажом, что привело к тяжелым финансовым потерям[1]. В таких ситуациях понимание и моделирование области управления багажом должно быть вашей основной задачей. А тратить время на оптимизацию пула соединений в базе данных будет ошибкой. В данном примере предметная область — это критически важная сложность задачи.

DDD проявляет себя лучше всего, когда система имеет дело с предметной областью, которую трудно понять. В таких ситуациях самыми насущными задачами являются понимание всей сложности предметной области и ее моделирование. Если вы не сумеете уловить всех тонкостей различных технических аспектов, ваша система получится не такой полезной, как хотелось бы. Но если не сможете разобраться в сложностях предметной области, то получите систему, которая делает не то, что нужно. В этом смысле предметная область обладает *критически важной сложностью*.

---

[1] Денвер в этом не одинок. Например, 5-й терминал аэропорта Хитроу в Лонгфорде (Англия) испытывал похожие проблемы.

Как показывает наш опыт, к этой категории относится большинство бизнес-приложений. Понимание предметной области и создание подходящей модели являются основой для решения задач бизнес-логики.

Вам может показаться, что предметная область — это что-то нетехническое. Но это не так. Иногда критически важная сложность относится к предметной области, а сама область является технической. Представьте, что вы пишете оптимизирующий компилятор. Он превращает исходники в хорошо оптимизированный машинный код, который можно выполнять, при этом он задействует локальные оптимизации, устраняет «мертвый» код, разворачивает подвыражения на этапе компиляции и т. д. Сложность здесь состоит не в повышении производительности чтения/записи файлов, а в применении всех этих оптимизаций таким образом, чтобы полученная программа делала то, что указано в исходном коде. Основные усилия должны быть направлены на строгое представление исходного кода и выполнение преобразований так, чтобы оптимизированная программа не меняла своей сути. Здесь критически важная сложность относится к предметной области, но сама область является технической!

Теперь попробуем разобраться, какое отношение это имеет к безопасности. Вам будет непросто получить все знания, необходимые для того, чтобы ваша система вела себя как следует в любых возможных ситуациях. Учитывая все странные случаи, которые могут возникнуть, эта задача будет довольно сложной даже при условии работы с нормальными, безопасными данными. Но все станет еще сложнее, если вам нужно обеспечить устойчивость к вредоносным данным. Кто-то может попытаться атаковать вашу систему, послав ей причудливый ввод, чтобы заставить ее сделать что-то неприятное. И даже в этом случае система должна ответить корректным и безопасным способом. Мы уже видели это на примере книжного интернет-магазина в главе 2. Никакая нормальная бизнес-процедура не может привести к добавлению в корзину антикниги в количестве –1. Тем не менее недобросовестный клиент может это сделать, чтобы манипулировать системой (в данном случае для уменьшения итоговой стоимости заказа).

Как показывает наш опыт, для обеспечения безопасности основное внимание необходимо уделять построению доменных моделей. Побочным эффектом этого подхода станет то, что можно будет избежать множества проблем с безопасностью, особенно тех, которые касаются бизнес-целостности. К тому же доменные модели в определенной степени защищают ваш код от некоторых технических атак.

Вам нужны доменные модели, которые способствуют стабильной и безопасной разработке. Эффективная модель должна:

- быть простой, чтобы вы могли сосредоточиться на ключевых моментах;
- быть строгой, чтобы служить основой для написания кода;
- обеспечивать глубокое понимание, чтобы сделать систему по-настоящему полезной;
- быть лучшим выбором с прагматической точки зрения;
- предоставлять язык, который можно использовать при обсуждении системы.

DDD не панацея: полезность этого подхода зависит от контекста. Существуют ситуации, в которых не стоит уделять основное внимание моделированию предметной области. Например, если вы пишете программное обеспечение для сетевого маршрутизатора, самым важным аспектом для вас будет пропускная способность ввода/вывода. В данном случае критически важная сложность лежит в технической плоскости. Но даже здесь необходимо подумать о том, не создает ли недоскональная доменная модель риски безопасности.

СОВЕТ

Критически важная сложность существует всегда. Вам нужно определить, к чему она относится: к техническим аспектам или к предметной области.

По нашему мнению, главным преимуществом моделирования предметной области является то, что оно поощряет более глубокое изучение предмета, без которого не обойтись. Овладеть жаргоном бизнес-специалистов не так уж сложно, и вы можете использовать тот же язык для составления перечня требований, который на первый взгляд выглядит прилично. Но если нет глубокого понимания данной области, в него могут проникнуть недоразумения, несоответствия и логические лазейки. Эти изъяны не дадут вам создать надежную систему, которая ведет себя правильно в нестандартных ситуациях, а в худшем случае станут причиной появления уязвимостей в безопасности. Работая над доменной моделью совместно со специалистами в предметной области, вы можете многому научиться.

3.1.1. Модель — это упрощенная версия реальности

Модель — это упрощенная версия реальности, из которой убрано все то, что вам не нужно. Например, когда вы регистрируете багаж в аэропорту, системе не нужно знать размер вашей обуви. А вот информация о весе чемодана может пригодиться. Чтобы вам было легче понимать и реализовывать систему, модель должна содержать только те сведения, которые вас интересуют, например вес багажа, но не размер обуви пассажира.

Следует понимать, что модель — это не диаграмма. Хотя во многих других контекстах *модель* может означать диаграмму определенного типа, как в случае с моделями отношений в проектировании баз данных или диаграммами классов в UML. Эти диаграммы являются представлениями моделей, но *сами* модели описывают упрощенное понимание реальности на концептуальном уровне.

ОБРАТИТЕ ВНИМАНИЕ

Модель — это не диаграмма, а определенный набор абстракций.

В DDD термин «модель» используется примерно в том же значении, что и в моделировании железнодорожного транспорта. При изготовлении моделей поездов

одним аспектам реальности уделяется большое внимание, а другие совершенно игнорируются (рис. 3.1). Понимание того, какие детали должны быть реалистичными, а какие можно исказить, — ключ к созданию как игрушечных поездов, так и моделей предметной области.

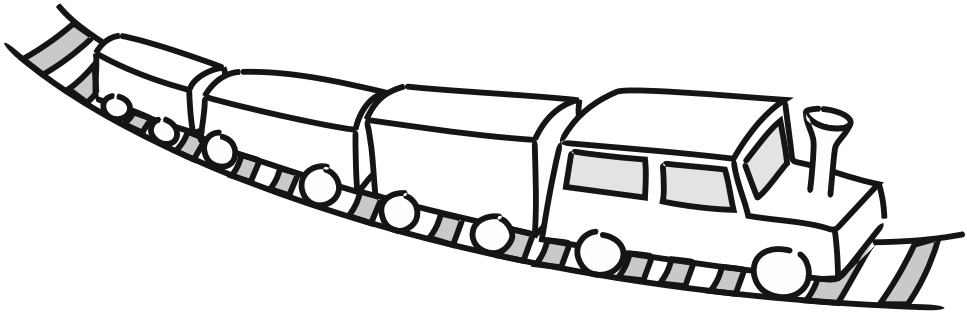

Рис. 3.1. Модель поезда выглядит как настоящий, подлинный поезд

На рис. 3.1 изображена модель поезда. Она выглядит как поезд и движется по рельсам, но настоящим поездом ее назвать нельзя. Мы считаем ее моделью, так как она сохраняет некоторые важные атрибуты, игнорируя остальные. Перечислим параметры, общие для игрушечного и настоящего поездов.

- *Цвет.* Мы считаем, что модель определенного поезда должна иметь тот же цвет, что и оригинал.
- *Относительные размеры.* Мы ожидаем, что пропорции будут соблюдены. Если в реальности высота дверей в два раза больше ширины, то же соотношение должно быть и в модели.
- *Форма.* Мы ожидаем, что модель поезда и ее детали будут иметь ту же форму, что и реальный поезд.
- *Движение.* Мы ожидаем, что модель поезда будет двигаться по рельсам таким же образом, как это делает настоящий поезд.

Назовем также некоторые атрибуты модели, точное воспроизведение которых мы считаем необязательным.

- *Материал.* Ничего страшного, если модель сделана из пластика или жести, а оригинал — из других материалов.
- *Абсолютный размер.* Если настоящие вагоны имеют длину 30 метров, то у модели они могут быть намного меньше, и это нормально.
- *Вес.* Модель намного легче, что вполне ожидаемо.
- *Механизм тяги.* У модели нет парового двигателя, она работает на электричестве.
- *Изгиб рельсов.* У модели рельсы могут изгибаться намного сильнее, чем в реальности, и это допустимо.

Удивительно, но найти различия между игрушечным и настоящим поездами намного легче, чем сходство. Тем не менее мы убеждены в том, что такая модель поезда правильная. Ей явно удалось вобрать в себя главные аспекты того, что мы понимаем под поездом.

Цвет, относительные размеры и движение — этого должно быть достаточно, чтобы понять, что перед нами поезд. Это три необходимых атрибута: если в модели они не соблюдены, мы не станем притворяться и называть ее поездом. Если же модели не удастся удовлетворить какие-то другие требования, можем это проигнорировать.

ОБРАТИТЕ ВНИМАНИЕ

Модель — это упрощенная версия реальности, которую мы все же можем воспринимать как корректное представление реальной вещи.

На этом мы завершаем экскурс в мир игрушек. Главная мысль, которую следует вынести из этого путешествия, состоит в том, что модель — это упрощенное представление чего-то реального. Это относится и к моделям, которые мы используем в системной разработке. При моделировании человека можно выбрать несколько атрибутов: имя, возраст, размер обуви и необязательное наличие домашнего питомца. Это, бесспорно, грубая, но все же модель (рис. 3.2).

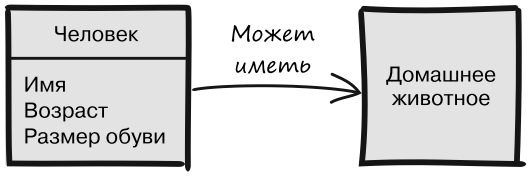

Рис. 3.2. Потенциальная модель людей и домашних животных

Модель — это упрощение, но она все же должна быть достаточно общей для того, чтобы охватывать различные интересные для вас вариации. В примере мы допускаем разные имена, возрасты и размеры обуви, а также наличие или отсутствие домашнего животного. Все это может проявляться в модели. Мы не различаем людей разного роста и не обращаем внимания на их прическу (рис. 3.3).

Эту модель можно представить множеством разных способов: использовать для этого обычный текст или проиллюстрировать ее с помощью разного рода диаграмм (например, сравните рис. 3.2 и 3.4). А еще применить код (псевдокод или настоящий язык программирования). Важно то, что ни одно из этих представлений не является моделью. В частности, за модель часто принимают диаграммы классов, но модели как таковые не тождественны своим представлениям. Модель — это концептуальное понимание того, что вы считаете существенным в процессе моделирования, — в данном случае это имя, возраст, размер обуви и домашний питомец.

Главное преимущество применения моделей в качестве сильно упрощенной версии реальности в том, что простые модели легче сделать строгими. Позже, когда вы будете создавать из них программное обеспечение, это будет важно.

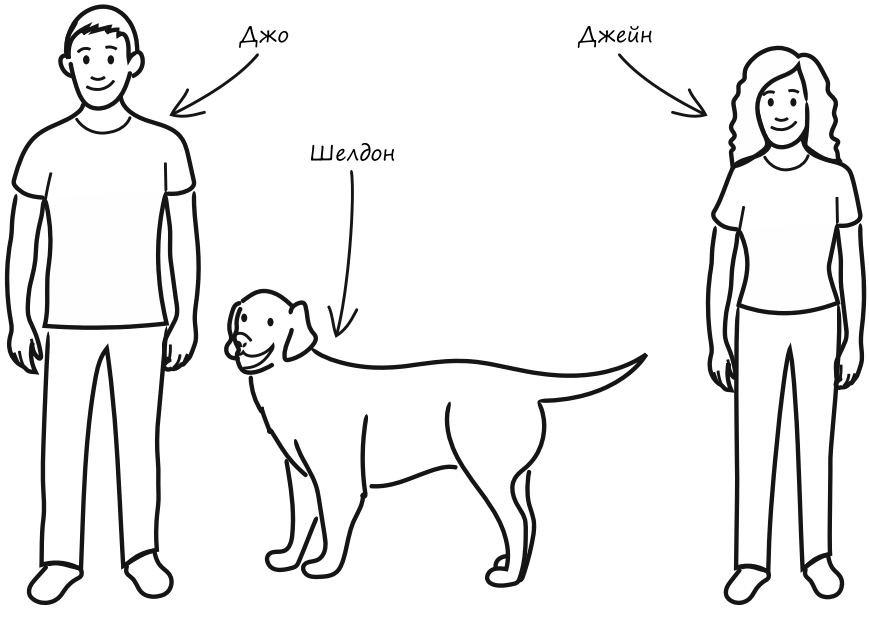

Рис. 3.3. Джо, 34 года, 9-й размер обуви, и его пес Шелдон; Джейн, 28 лет, 6-й размер обуви, без домашнего животного

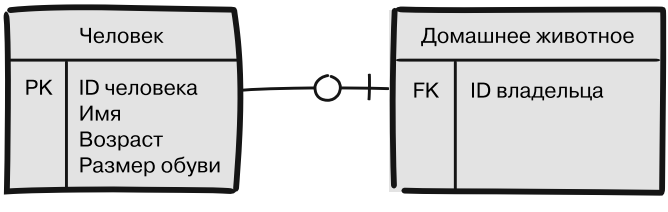

Рис. 3.4. Та же модель, но в другом представлении

3.1.2. Модели должны быть строгими

Доменная модель — это не просто упрощенная версия реальности: недостаток деталей компенсируется дополнительной строгостью. Слово «строгость» в данном случае используется в математическом смысле и означает «точность»: концепции, атрибуты, отношения и поведение должны быть однозначными.

Люди — очень сложные существа с множеством атрибутов и отношений. Решив сосредоточиться на имени, возрасте, размере обуви и домашних животных, вы потеряли много деталей. Но вместо этого получили точное определение того, что понимаете под человеком, и благодаря такой точности можете представить эту сущность в программном коде. Чтобы определить, какими деталями можно пожертвовать в угоду точности, нужно приложить много усилий. И если вы хотите сделать это как следует, придется обратиться к людям с глубоким пониманием предметной области.

ОБРАТИТЕ ВНИМАНИЕ

Тех, кто по-настоящему разбирается в той или иной теме, мы зовем специалистами в предметной области.

Написание программного обеспечения подразумевает сотрудничество между профессионалами двух разных профилей, которые должны продуктивно взаимодействовать. Речь идет о бизнес-специалистах и разработчиках. У каждой стороны есть особые потребности, удовлетворение которых является залогом создания отличного продукта. Бизнес-специалистам нужно иметь дело с терминологией, к которой они привыкли, а не с какой-то технической абракадаброй. Если они не узнают собственную предметную область, это будет означать, что вы их подвели.

Несколько терминов:

- *предметная область (домен)* — та часть реального мира, в которой что-то происходит (например, область управления багажом);
- *доменная модель* — отфильтрованная версия предметной области, в которой каждая концепция имеет определенное значение;
- *код* — закодированная версия доменной модели, написанная на языке программирования.

Одного лишь наличия знакомых слов в пользовательском интерфейсе или заголовках распечатанных отчетов недостаточно. Система должна вести себя так, чтобы бизнес-специалисты считали ее логичной, согласованной и понятной. Для этого доменная модель должна быть строгой. Если модель нестрогая и содержит двусмысленности, различные части системы могут вести себя по-разному.

Например, на экране в пункте регистрации может значиться «количество чемоданов», при выходе на посадку — «количество багажа», а на планшетах погрузчиков — «поклажа». Ситуация может усугубиться, если некоторые из этих понятий учитывают ручную кладь, а другие — нет. Когда работники аэропорта общаются между собой, каждый из них должен помнить, на какой экран смотрит его собеседник и следует ли при этом прибавлять/вычитать ручную кладь. Иногда возникают недоразумения, и багаж теряется. Система подводит компанию и выглядит нелогично даже для специалистов в предметной области.

ОСТОРОЖНО

Использование для описания концепции почти синонимов зачастую является признаком нестрогой модели.

Конфузы могут возникать и тогда, когда модель согласована с точки зрения терминологии, но имеет слишком мягкие ограничения и отношения. Во многих случаях это оказывается результатом задействования стандартной системы,

приспособленной к предметной области; так, к примеру, обычно применяют продукты ERP (enterprise resource planning — планирование ресурсов предприятия).

В 1940-х и 1950-х годах компьютеры впервые стали использовать в коммерческих целях: с их помощью планировали применение устройств и сырья на производстве, что было большим шагом вперед по сравнению с бумажным документооборотом и ручными процедурами. В 1980-х годах *планирование потребности в материалах* (material requirements planning, MRP) эволюционировало в *планирование производственных ресурсов* (manufacturing resources planning, иногда употребляют сокращение MRP II) и стало включать в себя финансы, персонал, маркетинг и другие так называемые ресурсы. Однако в исходной предметной области для производства продуктов по-прежнему использовались ресурсы из накладной на материальные средства. И, так как предприятия существуют разные, эти MRP имели очень гибкую конфигурацию.

В 1990-х годах эти процессы эволюционировали в системы ERP и начали применяться для планирования работы целых предприятий, а чтобы они могли действовать на любом предприятии в любой отрасли, их конфигурацию сделали еще гибче. Их часто описывали и продавали как *стандартные системы*, которые можно настроить для работы в любой предметной области. Однако они представляли собой те же системы управления материальными потоками. Это бизнес-направление успешно продавало такие системы для обработки жалоб клиентов, полицейских расследований и других совершенно разных сфер деятельности. К сожалению, успешная продажа системы вовсе не означала, что она приносила какую-то пользу. Если вы хотите сконфигурировать систему управления материальными потоками для проведения полицейских расследований, вам понадобятся очень неочевидные абстракции: полицейского можно представить в виде устройства, а отчет об ограблении можно считать грудой сырья, которая обрабатывается устройством (полицией) в ходе расследования.

Чтобы втиснуть одну предметную область в другую, вам нужно быть менее конкретными и точными. Результатом зачастую является общая *система управления объектами*, в которой любая сущность — это объект. Пользовательский интерфейс позволяет обновлять атрибуты объектов, но такая обобщенная модель несет в себе не так уж много информации о том, что эти объекты на самом деле представляют. Часто существует возможность указать любое сочетание атрибутов и отношений.

Такая нестрогая система, конечно же, склонна к ошибкам, а, как вы видели в примере с книжным интернет-магазином, недостаточная строгость может вызвать появление дыр в безопасности.

ОБРАТИТЕ ВНИМАНИЕ

Хорошая система заботится о потребностях как бизнес-специалистов, так и разработчиков. Она должна удовлетворять профессиональные нужды обеих этих групп.

Очевидно, что нам следует уделять внимание бизнес-специалистам. Они должны работать в привычной для себя предметной области, поэтому вам нужно выбрать

терминологию, которая им знакома. Если не удастся этого добиться, это станет большой ошибкой. Такой же большой, как неспособность удовлетворить нужды других профессионалов — разработчиков.

В конечном счете наши обязанности, как разработчиков, сводятся к написанию кода. И это математически строгий код — он говорит компьютеру, что делать в зависимости от имеющихся данных в соответствии с написанными нами правилами. Вот почему требуется строгость. И получить ее можно либо из разговоров со специалистами в предметной области, либо за счет заполнения пробелов с помощью обоснованных предположений.

Если мы скажем, что у большинства людей не больше одного домашнего животного, этого будет недостаточно. Разработчики должны знать, ограничено ли количество питомцев всего одним. Этот тот аспект нашей профессии, который требует определенного мужества. Вам нужно задать вопросы, чтобы сделать модель строгой и однозначной. Если вы спросите, может ли быть больше одного домашнего животного, в ответ можете услышать: «О, ну это очень редкое явление». В результате можете либо позволить указывать список животных, либо удовлетвориться тем, что животное может быть всего одно. В первом случае система способна стать сложнее необходимого и рано или поздно столкнется с каким-то необычным сочетанием атрибутов. Во втором случае вы запретите указывать больше одного домашнего питомца и несколько месяцев спустя ощутите на себе негодование клиентов (которые, возможно, достались вам при покупке другой компании), у которых их два и больше. Что еще хуже, бизнес-специалисты могут сделать вас козлом отпущения, лукаво заявив: «Мы же говорили, что это может случиться», хотя вы всего лишь сделали резонное предположение, чтобы не усложнять систему. У вас должна быть возможность принимать решения, чтобы продвигать разработку.

Чтобы не попасть в эту ловушку, следует активно интересоваться тем, какой должна быть модель: «Должны ли мы разрешить добавление нескольких домашних животных или лучше ограничиться одним?» Решение о том, следует ли учитывать потребности необычных клиентов, относится к бизнесу, а не к технологиям. Если множественные домашние животные не поддерживаются вашей системой, их придется обрабатывать с помощью отдельной ручной процедуры. Но и поддержка большого разнообразия тоже имеет свою цену. Возможность работы с все более общими моделями выглядит соблазнительно, но рано или поздно все ваши сущности будут соединены между собой отношениями вида «многие ко многим». В долгосрочной перспективе это не приведет ни к чему хорошему. Последствия использования общей модели сложно предвидеть и понять.

Представьте, что у вас есть функция, которая позволяет клиентам обмениваться домашними животными. Если при этом у одного человека может быть несколько четвероногих, вам придется переосмыслить эту возможность. Означает ли это, что клиенту A достаются все животные клиента B и наоборот? Или животные обмениваются по одному? Если не отразить в модели бизнес-область, можно подвести бизнес-специалистов. Если же создать недостаточно строгую модель, можно подвести разработчиков.

ОБРАТИТЕ ВНИМАНИЕ

Хорошая модель должна отражать бизнес-область и при этом быть строгой. Наличие строгой модели означает, что в конечном счете ее можно будет взять за основу для написания кода.

Проектируя программное обеспечение, вы принимаете похожие решения — создаете простое представление сложного явления. Рассмотрим пример объектно-ориентированного кода, который обычно используют в школьных учебниках. Это очень поверхностное описание человека, в котором игнорируется множество атрибутов и отношений:

```
class Person {                          ← Модель концепции предметной
    private String name;                  области «человек», представленная в коде

    private int age;
    private int shoeSize;
    private Animal pet;
    void growOlder() {
        this.age++;
    }
    void swapPetWith(Person other) {
    ...
    }
}
```

Здесь нет огромного количества свойств и операций, характерных для человека. Модель сведена к четырем атрибутам, которые важны в конкретном контексте. У нее есть назначение — сфера поведения, которую вы хотите описать. На первый взгляд отказ от деталей делает систему беднее, но взамен мы получаем кое-что очень важное — возможность быть точными.

Человек — сложное существо со сложными связями. Но в нашей модели класс `Person` — это нечто с именем, возрастом, размером обуви, домашним животным и способностью становиться старше. И все. Это то, что мы понимаем под термином *«человек»*. Теряя детали, мы приобретаем точность.

3.1.3. Модели вбирают в себя глубокое понимание предметной области

Конечно, предыдущий пример моделирования человека получился до смешного простым. Реальные проблемы намного сложнее, как в случае с обработкой багажа в аэропорту. Четкое понимание того, что именно охватывает доменная модель, дается не так легко, как многие могли бы подумать. На самом деле знания, которые вам нужно охватить, даже глубже тех, которыми пользуются специалисты в предметной области в повседневной работе, когда решают отдельные задачи. Причина этого в том, что понимания вам должно быть достаточно не просто для того, чтобы работать в конкретной предметной области, но для создания компьютерной системы. Сравним это с ездой на велосипеде.

Большинство из нас являются специалистами по езде на велосипеде в том смысле, что мы не обдумываем каждое свое движение[1]. В этом легко убедиться, если проехаться на велосипеде в непростых условиях: по неровной дороге, в ветреную погоду и, возможно, даже с большим пакетом в одной руке. Это требует мастерства. Сравните это с трудностями, с которыми сталкивается ребенок, который только учится кататься по гладкой поверхности в солнечный летний день. С таким мастерством сравнимы знания специалистов в предметной области — они разбираются в своей сфере деятельности. Например, специалист по доставке знает, как выстраивать маршруты для грузовых контейнеров даже в сложных условиях, например, когда контейнер по ошибке выгрузили с корабля и в ближайшее время никаких морских рейсов в том же направлении не предвидится. Специалист в предметной области способен уладить даже каверзные ситуации, рассматривая их по очереди.

К сожалению, написание программной системы требует еще более глубокого понимания. Вы не можете находиться «на объекте», оценивая возникающие проблемы и импровизируя, чтобы их решить, — все это недоступная вам роскошь. Вы пишете программу, которая должна все это делать без вашего присутствия. Более подходящая аналогия здесь скорее не обучение ребенка езде на велосипеде, а создание робота-велосипедиста.

При разработке такого робота понимание езды на велосипеде должно быть намного глубже, чем у большинства специалистов, включая профессиональных велокурьеров или тех, кто занимается велосипедным мотокроссом. Например, как повернуть направо, когда вы едете на велосипеде? Подумайте об этом несколько секунд — вы, наверное, делали это тысячу раз. Большинство людей ответит не задумываясь: «Я потяну за правую ручку руля». К сожалению, из-за центробежной силы вы наклонитесь влево и упадаете на асфальт[2].

На самом деле вы подсознательно поворачиваете руль влево, что заставляет вас на миг отклониться вправо. Через несколько миллисекунд наклон достигнет подходящего угла, после чего вы повернете руль вправо и выполните маневр. Наклон вправо должен быть ровно таким, чтобы компенсировать центробежную силу, он позволит выполнить безопасный и стабильный поворот (рис. 3.5). Вы делаете это, не задумываясь и не понимая всех тонкостей кинематики. Если хотите создать робота-велосипедиста — это тот уровень, на котором вы должны владеть данной темой.

История с роботом несет как плохие, так и хорошие новости. Плохая новость заключается в том, что в голове у специалиста в предметной области нет готовой, *настоящей* модели. Вы не сможете получить у него ответы на все свои вопросы.

---

[1] В дрейфусовской модели приобретения навыков людей с умениями такого уровня называют экспертами и мастерами. Можете почитать книгу A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition (University of California, Berkeley, 1980), написанную братьями Стюартом и Хьюбертом Дрейфусами.

[2] Центробежные силы действительно существуют, даже если ваш учитель по физике говорил обратное. Речь идет о фиктивной силе, которая наблюдается во вращающейся системе отсчета, такой как поворачивающий велосипедист. Герберт Голдштейн написал замечательную книгу по кинематике под названием Classical Mechanics (Addison-Wesley, 1951).

Но есть и хорошая новость: совместная работа над моделью со специалистами в предметной области может быть веселой и полезной. Это процесс постепенного исследования разных моделей с выбором той из них, которая подходит для решения имеющихся задач.

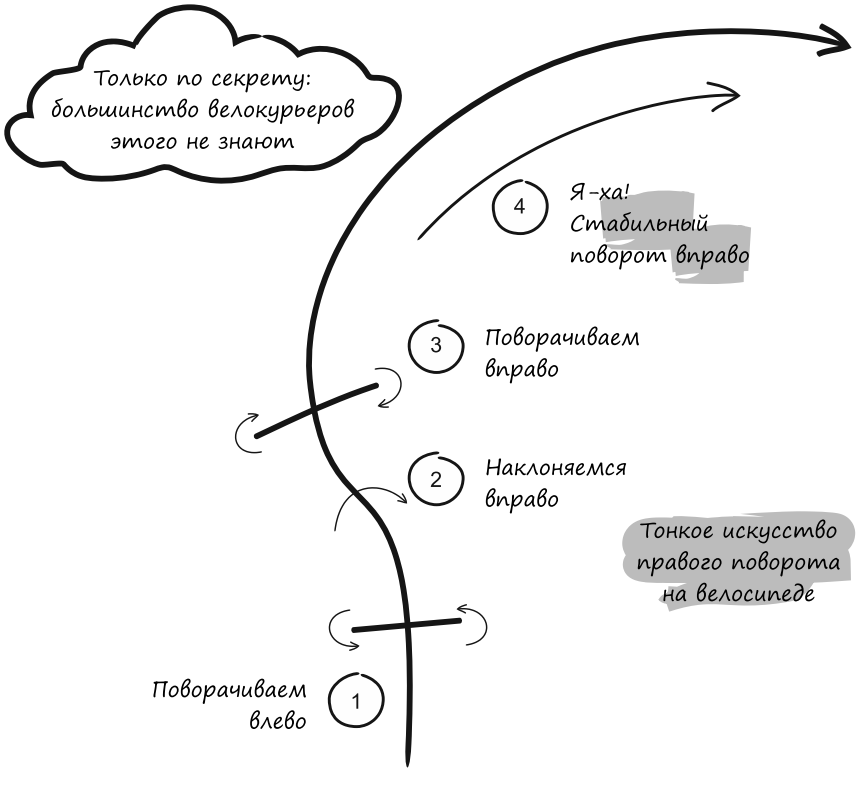

Рис. 3.5. Чтобы создать робота-велосипедиста, требуется глубокое понимание того, как поворачивать направо

СОВЕТ

Лучшие модели получаются при взаимодействии разработчиков и специалистов в предметной области — это постепенный и многошаговый процесс.

3.1.4. Модель не создают, а выбирают

Один из распространенных мифов о моделировании, которого придерживаются многие, предполагает существование в голове у специалиста в предметной области идеальной модели. Это не так. Создание модели подразумевает сознательный выбор из множества вариантов: модель должна соответствовать вашим потребностям, которые определяют ее назначение.