ГЛАВА 1

Проблемы коммуникации

Индустрия разработки ПО подошла к интересному рубежу. Распределенные по всему миру команды создают на различных языках программирования системы огромного масштаба, архитектура которых варьируется от монолитной до составленной из десятков распределенных микросервисов. Подходы Agile и Lean стали общепринятыми, и даже самые консервативные организации проверяют свои идеи с помощью MVP (минимально жизнеспособного продукта), чтобы быстро получить обратную связь. Благодаря методам вроде автоматизированного тестирования и непрерывной доставки в сочетании с возможностями облачных вычислений и невероятным ускорением, которое обеспечивает генеративный искусственный интеллект (GenAI), разработка масштабных систем теперь по плечу организациям любого размера. Но кое-чего все же недостает.

Попросите строителя визуализировать архитектуру здания, и вы получите набор профессиональных чертежей: генплан, поэтажные планы, виды фасадов, разрезы с детальной проработкой отдельных фрагментов. Попросите разработчика визуализировать архитектуру ПО — и, скорее всего, увидите лишь хаотичную мешанину из прямоугольников и линий.

На своих семинарах, которые я провожу с середины 2000-х годов, я просил тысячи разработчиков со всего мира нарисовать диаграммы архитектуры ПО. Будь то «архитектурная ката», где нужно спроектировать

систему, или мастер-класс, на котором разработчики рисуют диаграммы, описывающие архитектуру своих текущих проектов, — результат всегда один: наспех сделанный набор из «квадратиков и стрелочек», понятный только его авторам. Как вы увидите далее, двусмысленность таких диаграмм серьезно мешает разработчикам понимать друг друга и задачи проекта.

Но актуальны ли на сегодняшний день диаграммы архитектуры ПО? Да, и даже более чем когда-либо, учитывая стремление к быстрой доставке и сложность современных решений. Кроме высоких требований к технической подготовке разработчиков, сама разработка ПО, которая является коллективной деятельностью, требует, чтобы у всех участников было общее понимание проекта. Диаграммы — незаменимый инструмент для формирования единого видения архитектуры ПО внутри команды и за ее пределами, для масштабирования команды (например, при адаптации новичков), а также для сохранения знаний, которые понадобятся в будущем при обслуживании системы. Общий контекст жизненно важен и для внедрения искусственного интеллекта (ИИ) в рабочие процессы, чтобы ИИ-агенты могли вносить изменения, не нарушая существующую архитектуру.

Эта глава начинается с обзора проблем, связанных с диаграммами архитектуры ПО, и анализа того, как индустрия пришла к текущему положению дел. В следующих главах я предложу решение — модель C4, простой и удобный для разработчиков подход к визуализации архитектуры ПО.

Что такое архитектура ПО

Прежде чем обсуждать диаграммы архитектуры ПО, определим, что представляет собой эта архитектура. По данной теме накоплен огромный пласт знаний, так что мой рассказ будет заведомо неполным, но для целей книги достаточным.

Грейди Буч (Grady Booch), один из создателей унифицированного языка моделирования UML (Unified Modeling Language), в *Handbook of Software Architecture* (https://oreil.ly/lz4m3) писал: «Архитектура — это совокупность значимых проектных решений, которые определяют структуру

и функции системы, где значимость измеряется стоимостью внесения изменений». Мне нравится это определение, так как оно помогает отделить архитектурные аспекты от прочих. На своих семинарах я прошу участников назвать самые сложные и дорогостоящие изменения в их ПО. Как правило, ответы сводятся к следующему:

- *технологии* — выбор языков программирования, библиотек, фреймворков, сред развертывания и т. д.;
- *элементы* — способ разделения ПО на исполняемые блоки (например, монолиты и микросервисы, организация кода по слоям или по функциональности) и способ хранения и структурирования данных (схемы, форматы);
- *взаимосвязи* — зависимости и взаимодействия между элементами (синхронный/асинхронный обмен информацией, форматы данных, протоколы).

На мой взгляд, диаграммы архитектуры ПО должны иллюстрировать ключевые решения, касающиеся технологий, элементов и их взаимосвязей. Все эти аспекты важны, поскольку они отражают как проектные решения, принятые командой, так и внешние ограничивающие факторы (проблемная область, технологические ограничения и отношения с поставщиками, кадры, существующие системы, протоколы интеграции, практики эксплуатации и т. д.).

Популярные проблемы при создании диаграмм

Вновь обратимся к семинарам. Участникам предлагается спроектировать для банка небольшую систему оценки финансовых рисков и нарисовать диаграммы, отражающие архитектуру ПО для этого решения. Система должна импортировать данные из двух источников: TDS (Trade Data System — система торговых данных) и RDS (Reference Data System — система справочных данных), объединять эти наборы данных, рассчитывать риски и формировать отчет в формате Microsoft Excel для внутренних бизнес-пользователей. Некоторым пользователям разрешается изменять параметры расчетов риска.

Набор функциональных требований системы и атрибутов качества (производительность, масштабируемость) намеренно ограничен, чтобы решения были максимально простыми, а упражнение можно было бы выполнить за 60–90 минут. Это упражнение по проектированию ПО, однако основное внимание уделяется именно диаграммам. В идеале я ожидаю увидеть диаграммы архитектуры ПО, отражающие результат процесса проектирования, пройденного группами для выполнения требований, и демонстрирующие взвешенный подход к выбору технологий, элементов и их взаимосвязей. Но на практике все иначе: диаграммы получаются очень разными. Одни — слишком абстрактные, другие — излишне детализированные. Одни показывают статическую структуру, другие — динамику и поведение. Некоторые показывают выбор технологий, а большинство — нет.

Как правило, диаграммы страдают от одной или нескольких следующих проблем:

- нотация (формы, цветовое кодирование, значки, стили линий) не объясняется или противоречива;
- назначение и смысл элементов неоднозначны;
- отсутствуют связи между элементами;
- связи между элементами не подписаны;
- используются общие термины, например «бизнес-логика»;
- не объясняются акронимы и аббревиатуры;
- не указан выбор технологий;
- смешаны уровни абстракции.

Более того, при создании набора диаграмм эти проблемы усугубляются:

- нотация на разных диаграммах не совпадает;
- названия элементов на разных диаграммах не согласуются;
- логическая последовательность чтения диаграмм неясна;
- отсутствует четкий переход между диаграммами.

Для иллюстрации перечисленных проблем рассмотрим несколько примеров. На рис. 1.1 показан набор «высокоуровневых логических строительных блоков», из которых строится решение, и связи между ними. К сожалению, стрелки не подписаны, и мы практически не получаем информации о назначении этих связей. Пояснение «Безопасность» на блоке слишком расплывчатое.

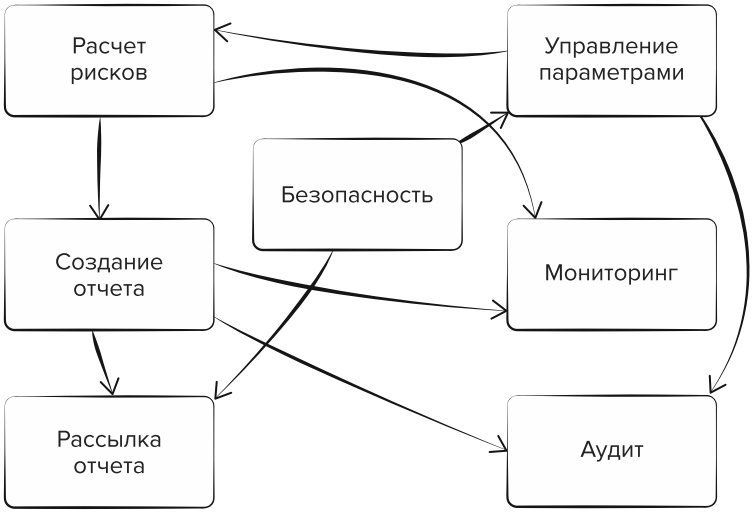

Рис. 1.1. Пример диаграммы с семинара, которая показывает «логическое представление» решения, с неподписанными связями

Следующий пример (рис. 1.2) демонстрирует еще один распространенный стиль диаграмм. Большая часть представленной на ней информации носит слишком общий характер и описывает практически все типы программных систем, которые создают инженеры-программисты.

Большой прямоугольник RS обозначает границу проектируемой системы управления финансовыми рисками (Financial Risk System), а вложенные прямоугольники — ее декомпозицию на меньшие «строительные блоки». Проблема в том, что блоки и связи подписаны слишком общими терминами: «Транспорт», «Ошибка», «БД». В частности, надпись на блоке в центре «Бизнес-логика» совершенно неинформативна. Если переименовать его в «Калькулятор финансовых рисков», то это даст хотя бы некоторое представление о его назначении в предметной области.

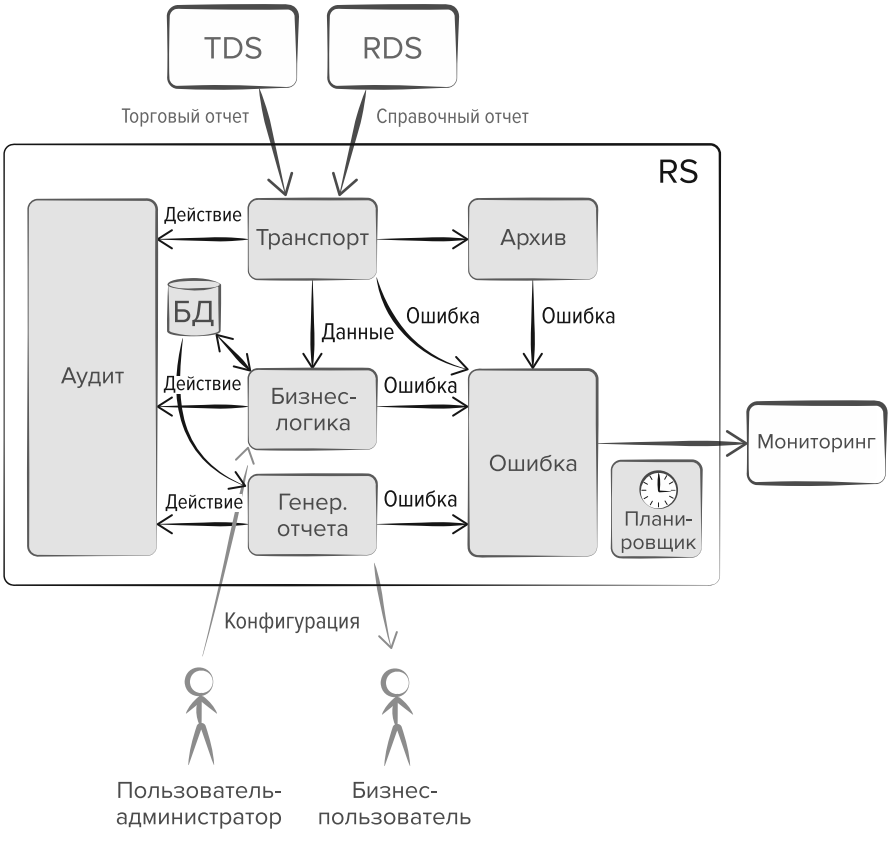

Рис. 1.2. Пример диаграммы с семинара, демонстрирующей обобщенное описание решения

Наконец, на рис. 1.3 показана общая структура решения, состоящего из веб-приложения, пакетного процесса и хранилища данных.

Мне нравится, что на диаграмме показаны зоны ответственности каждой части решения — это позволяет понять, как распределена функциональность. Однако выбор технологий отсутствует. На вопрос «Почему?» обычно дают один из следующих ответов:

- «Решение [система финансовых рисков] простое, и его можно построить на любой технологии»;
- «Мы не хотим навязывать решение разработчикам»;
- «Это деталь реализации».

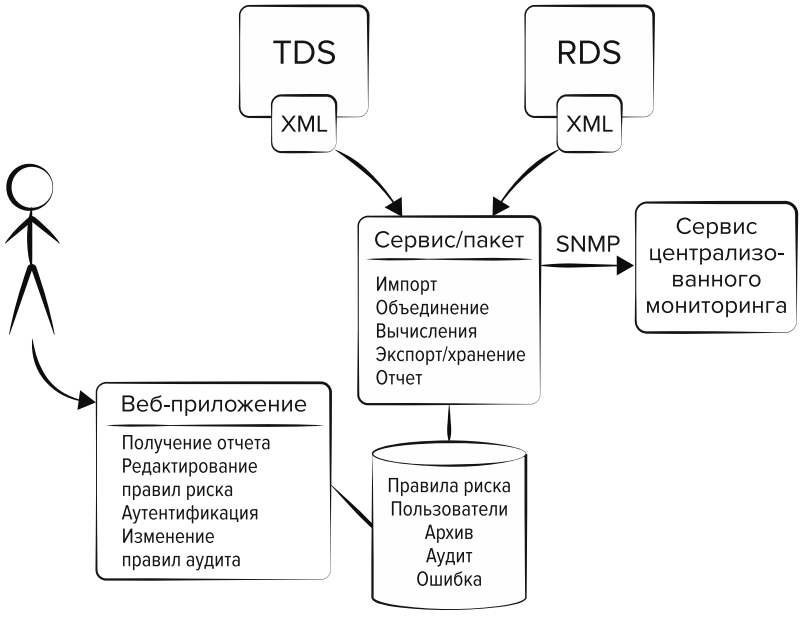

Рис. 1.3. Пример диаграммы с семинара: показаны зоны ответственности, но отсутствует выбор технологий

По ряду причин, которые мы обсудим далее, я настоятельно рекомендую указывать технологии на диаграммах архитектуры ПО.

Эти примеры типичны для моих семинаров, но такие проблемы с диаграммами на практике встречаются повсеместно. Поищите в сети изображения по запросу «диаграмма архитектуры ПО», и вы найдете множество подобных схем из «квадратиков и стрелочек» с теми же самыми недостатками.

Упражнение по созданию диаграмм обычно выполняется группой из 2–4 человек. Но участие людей с разным опытом и подготовкой не облегчает выполнения упражнения, а усложняет: масса времени уходит на споры, как лучше реализовать решение. Опросы участников о том, что было самым сложным при создании диаграмм, выявили, что визуальной коммуникации в области архитектуры ПО никто предварительно не обучался. Я постоянно слышу одни и те же вопросы:

- «Какие типы диаграмм нам рисовать?»;
- «Какую нотацию нужно использовать?»;
- «Какой уровень детализации выбрать?»;
- «Для какой аудитории эти диаграммы?».

Мой опыт показывает, что как в индустрии разработки программ, так и в академической среде разработчиков редко обучают эффективно моделировать, визуализировать и описывать архитектуру ПО. Эта тема все еще считается нишевой, несмотря на отрицательное воздействие слабой коммуникации на команды разработчиков.

Что случилось с UML

В сущности, в сложившейся ситуации нет ничего удивительного. В отличие от строительной отрасли в разработке ПО нет единого стандарта для осмысления, описания и визуализации архитектуры. И хотя у нас есть универсальный язык моделирования — UML, — он, по-видимому, не помогает.

Я окончил университет в 1996 году и вскоре после этого прошел недельный курс по UML. Я устроился разработчиком в консалтинговую компанию. Мне приходилось регулярно писать документацию для наших клиентов, чтобы они могли использовать и обслуживать созданное нами ПО. Мы часто применяли UML, создавая самые разные диаграммы с помощью различных инструментов. Эта практика получила еще большее распространение в эпоху доминирования методологии Rational Unified Process (RUP) с конца 1990-х по начало 2000-х. Иными словами, я хорошо знал UML, имел большой опыт работы с ним, да и клиенты часто просили его применять. Но в середине 2000-х все изменилось: популярность UML начала стремительно убывать.

На мой взгляд, этому поспособствовало несколько факторов. Ранние версии UML 1.x были лаконичны, но UML 2.x разросся, отчасти из-за внедрения MDA (Model Driven Architecture) — модельно-ориентированного подхода к разработке программного обеспечения. Текущая

спецификация UML (v2.5.1 на время написания книги) — это почти 800-страничный PDF-документ (https://oreil.ly/LW79L). Хотя для работы с UML не нужно знать *все о языке*, огромный объем стал серьезным барьером для многих разработчиков.

Хотя в UML есть диаграммы для разных уровней абстракции, исторически фокус сместился на диаграммы классов, которые в основном один в один отражают код. Для проектирования или документирования небольшой кодовой базы из десятка классов UML-диаграммы классов могут быть полезны. Однако для систем с сотнями и тысячами классов они быстро становятся «неподъемными». В начале 2000-х были инструменты для реверс-инжиниринга (построения UML-диаграмм классов из кода), для генерации кода из UML-диаграмм и даже инструменты с полным циклом «код — диаграмма — код». Все они были, как правило, ненадежны и работали с серьезными ограничениями.

Сегодня разработчики редко рисуют UML-диаграммы классов перед созданием кода. Они предпочитают комбинировать такие методы, как TDD (Test-Driven Development — разработка через тестирование), рефакторинг и даже генеративный ИИ, которые доступны в современных инструментах разработчика. Исторически фокус UML избыточно смещался на уровень, близкий к коду, и сегодня многими воспринимается как «слишком детализированный». К тому же недостаточно внимания уделялось практическим рекомендациям по применению UML, особенно на высоких уровнях абстракции, для повседневного моделирования корпоративных систем. Как мы увидим, эффективное использование UML — это гораздо больше, чем простое применение нотации.

Свою роль сыграло и широкое распространение Agile. Опубликованный в 2001 году Manifesto for Agile Software Development («Манифест гибкой разработки программного обеспечения», https://agilemanifesto.org) ознаменовал переход от громоздких, ориентированных на документацию подходов с длинными циклами обратной связи к более итеративным и адаптивным моделям работы. Принцип «Работающее ПО важнее исчерпывающей документации» многие неверно истолковали как призыв не писать ее вовсе. В спешке внедряя Agile, многие команды вместе с водой выплеснули и ребенка. Диаграммы и документация прочно

ассоциировались с традиционными подходами, основанными на предварительном планировании. Команды разработчиков отказались не только от этих старых подходов, но, к сожалению, и от диаграмм и документации. Во многих командах, с которыми я работал, документация была крайне скудной. Я до сих пор регулярно слышу ложное утверждение, что «Agile не требует документации».

Команды разработчиков, понимающие ценность диаграмм и документации, отказались от UML в пользу более легких, но, к сожалению, бессистемных (ad hoc) решений. Исходя из моих разговоров с тысячами разработчиков из 40 стран, можно предположить, что UML использует малая часть нашей индустрии — думаю, менее 10 %. Есть исключения (например, при разработке встраиваемых систем UML применяют чаще, чем для корпоративных веб-приложений), но общая тенденция неизменна. Даже те, кто заявляет об использовании UML, обычно ограничиваются диаграммами классов и последовательности (sequence diagrams).

В академической среде тоже заметили, что индустрия разработки ПО не слишком активно использует UML, поэтому некоторые университеты сократили количество часов на его изучение. Недельный курс по UML в рамках трехлетней программы обучения — обычное дело. В итоге выпускники получают ограниченное представление о UML и приходят в команды, где его не применяют. Через пять лет вчерашние студенты становятся техническими руководителями, и им требуется нарисовать несколько диаграмм. Станут ли они использовать UML? Вряд ли, ведь у них нет опыта работы с ним. Ситуацию могут изменить лишь серьезные преобразования в самом UML или его восприятии.

Впрочем, UML не решает основных, рассмотренных нами проблем с диаграммами. Чтобы понять почему, взгляните на рис. 1.4 — это версия рис. 1.1, который я перерисовал с использованием нотации UML.

Здесь показаны компоненты UML и отношения зависимости между ними. Дает ли эта диаграмма больше информации, чем «квадратики и стрелочки» на рис. 1.1? Едва ли. Можно ли назвать ее хорошей? На самом деле нет. Ее можно было бы улучшить, добавив предоставляемые

и требуемые интерфейсы для каждого компонента или хотя бы описания зависимостей. Но даже тогда диаграмма осталась бы расплывчатой, поскольку она не предоставляет информации о поведении компонентов. Добавление нотации UML не помогает само по себе. Большинство рассмотренных проблем слишком фундаментальны, чтобы их можно было устранить с помощью одной лишь стандартной нотации.

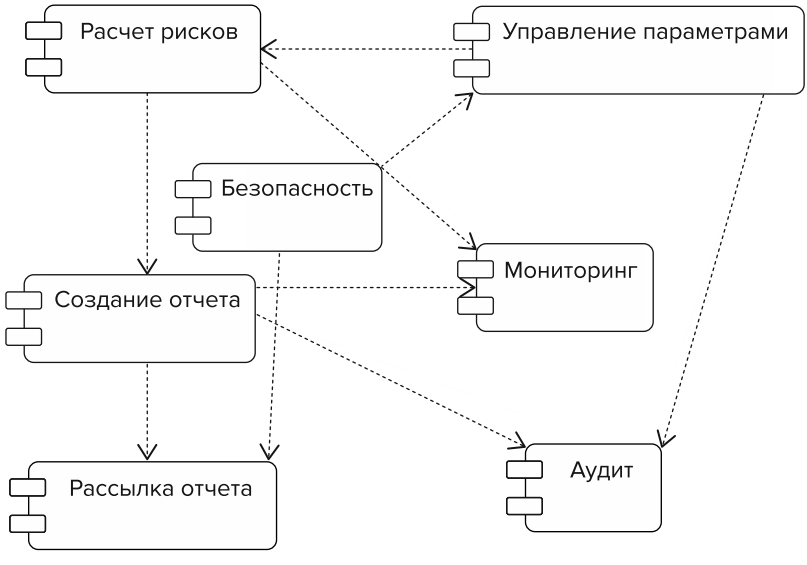

Рис. 1.4. Вариант диаграммы с рис. 1.1, в котором вместо произвольных «квадратиков и стрелочек» используется нотация UML

Детализация технологий на диаграммах

Вы, возможно, заметили, что в примерах почти нет сведений о выборе технологий. Это обычное явление. Нужно ли указывать на диаграммах архитектуры ПО технологии? Безусловно, да.

К сожалению, бытует мнение, что диаграммы архитектуры ПО должны быть высокоуровневыми и логическими, то есть показывать разбивку программной системы на функциональные строительные блоки, но не способ их реализации. Я с этим не согласен и предлагаю разобраться в причинах появления такой точки зрения.

До повсеместного распространения Agile нормой циклов поставки ПО считался каскадный подход («водопад»). Он предполагал, что сначала выполняется этап проектирования/архитектуры, за которым следует отдельный этап реализации/написания рабочего кода. Методологии того времени часто предписывали поручать эти этапы разным командам: архитекторам и разработчикам. По моему опыту, в команду архитекторов нередко входили люди, далекие от программирования, или бывшие разработчики, которые не писали код уже несколько лет. Эта команда отвечала только за создание высокоуровневых логических диаграмм архитектуры — выбор деталей реализации не входил в их обязанности даже при наличии необходимых компетенций. Эти диаграммы передавались команде разработчиков для воплощения в жизнь. Я считаю, что именно это исторически сложившееся разделение труда и привело к разграничению на «логические» и «физические» диаграммы, а в итоге — к распространенному сегодня убеждению, что выбор технологий в диаграммы архитектуры ПО включать не следует.

Современная разработка ПО строится на гораздо более тесном сотрудничестве: команды стали меньше, циклы доставки — короче, решения принимаются децентрализованно, а «пишущие программный код архитекторы» (hands-on coding architects) встречаются все чаще. В таких условиях теряют смысл создание и сопровождение нескольких наборов архитектурных диаграмм: для высокоуровневого логического представления и детальной физической реализации. Тем более что высокоуровневые логические диаграммы часто слишком абстрактны и не содержат информации о технологиях, элементах и связях, изменять которые в будущем окажется слишком дорого. Иными словами, они не отражают значимых решений.

Во многих организациях я слышу: «Мы не хотим указывать разработчикам, что делать». Решение здесь простое: привлекайте разработчиков к проектированию архитектуры, вместе выбирайте общую структуру и технологии для ее реализации. Архитектура — это совокупность решений, которые трудно или дорого изменить. Сюда входят декомпозиция программной системы на блоки (элементы), их взаимодействие (взаимосвязи) и выбор технологий для реализации. Проектирование ПО — это принятие всех этих решений с сохранением баланса между возможностями команды и ограничениями среды. Выбор

технологий — не просто деталь реализации, особенно если их замена в будущем обойдется дорого.

Выбор технологий помогает связать излишне идеалистичный и концептуальный проект ПО с реальностью и донести до всех участников всю полноту проектных решений, а не их часть. Кроме того, указание технологий на диаграммах, особенно в процессе проектирования ПО, гарантирует, что диаграммы рисуют компетентные люди — те, кто в этих технологиях разбирается.

Неоднозначность порождает домыслы

Если разобраться с проблемами нотации, примеры диаграмм станут лучше, но не будет решена главная проблема: во многих случаях из этих диаграмм вообще невозможно понять, какое именно решение предлагается. Я бы не советовал делать то, что сейчас скажу! Однако ни одну из показанных диаграмм нельзя отдать команде разработчиков и ожидать, что они создадут по ней готовый продукт. Попросту говоря, в этих диаграммах содержится крайне мало информации о решении.

Простейший способ проверить, имеет ли диаграмма смысл, — дать ее кому-нибудь и попросить истолковать без ваших комментариев. Я убежден, что диаграммы в определенной степени *должны говорить сами за себя*. Любые пояснения должны их дополнять, а не объяснять. Однако на моих семинарах часто звучит:

- «Мы просто расскажем, что изображено на диаграммах»;
- «Тут не все понятно, но мы объясним на презентации».

Расчет на то, что диаграмма будет сопровождаться устными пояснениями, создает разрыв между тем, что запечатлено на бумаге, и тем, что остается в головах. Ценность диаграмм, требующих разъяснений, невелика, особенно в качестве долговечной документации.

Неоднозначность диаграмм порождает домыслы — верные или нет, зависит от того, насколько опыт читателя сопоставим с опытом автора

диаграммы. Я не раз наблюдал, как три человека увлеченно проектируют решение, рисуя на диаграмме прямоугольники — будущие части системы.

Спустя час мы переходим от рисования диаграмм к их разбору. И только теперь слышим следующее.

> *Участник 1:* Эти прямоугольники — компоненты монолита, ведь так?
>
> *Участник 2:* Ой, я думала, мы проектируем микросервисную архитектуру.
>
> *Участник 3:* Да, я тоже решил, что это микросервисы.

Такое не раз случалось на моих семинарах, особенно если в группу входили люди с различным опытом, из разных команд или компаний. Это показатель того, что в ходе проектирования не удалось выработать общего видения и создать единый контекст: каждый смотрел на диаграмму через призму собственного мировосприятия.

Резюме

Чтобы понять, как делать что-то правильно, важно знать, чего делать не следует. В разработке ПО верные и неверные подходы часто называют *паттернами* и *антипаттернами*. Понимание того, что следует делать, а что — нет, — ключ к мастерству.

В этой главе мы обозначили проблему: многие разработчики не умеют создавать качественные диаграммы архитектуры ПО. Почему ее важно решить? В условиях современной доставки ПО это неумение приводит к тому, что команды теряют способность объяснить друг другу, что они создают. Неудивительно, что таким командам зачастую недостает технического лидерства, единого направления и согласованности в действиях. Чтобы всем вместе двигаться к общей цели, эту цель нужно уметь донести до каждого, а для быстрой и гибкой разработки — делать это эффективно.

Иными словами, скорость невозможно обеспечить без эффективной коммуникации. Качественная коммуникация:

- помогает каждому понять общую картину проекта и его место в ландшафте организации;
- формирует у команды разработчиков единый контекст и общее видение;
- служит разработчикам «картой» для навигации по исходному коду;
- становится отправной точкой для обсуждений, в которых затрагиваются сложные темы: новые функции, технический долг, анализ рисков, моделирование угроз, архитектурные ревью (проверки) и т. п.;
- ускоряет адаптацию новых разработчиков;
- дает возможность объяснить суть проекта нетехническим специалистам и другим командам.

В главе 2 я представлю свое решение этих проблем — модель C4 (C4 model) для визуализации архитектуры ПО.