Майкл Л. Перри

# Искусство неизменяемой архитектуры

## Теория и практика управления данными в распределенных системах

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Майкл Л. Перри

# Искусство неизменяемой архитектуры:
## теория и практика управления данными в распределенных системах

# The Art of Immutable Architecture

## Theory and Practice of Data Management in Distributed Systems

**Michael L. Perry**

Apress®

# Искусство неизменяемой архитектуры:
## теория и практика управления данными в распределенных системах

**Майкл Л. Перри**

ДМК Пресс
издательство
Москва, 2022

**УДК 004.4**
**ББК 32.971.3**
**П26**

**Майкл Л. Перри**
**П26** Искусство неизменяемой архитектуры: теория и практика управления данными в распределенных системах / пер. с анг. С. В. Минца; науч. ред. В. С. Яценков. – М.: ДМК Пресс, 2022. – 388 с.: ил.

**ISBN 978-5-93700-111-5**

Эта книга раскрывает преимущества использования неизменяемых объектов в распределенных системах. Вы узнаете о том, почему важна неизменяемость, исследуете пространство альтернатив и аспекты исторического моделирования. Затем ознакомитесь с математическими основами неизменяемости и увидите, как применять эти знания для анализа систем, построения машин состояний и соблюдения правил безопасности. В завершение будут рассмотрены компоненты компьютерной системы и их использование в неизменяемой архитектуре.

Издание предназначено для архитекторов программного обеспечения и опытных разработчиков, а также аналитиков бизнес-систем.

УДК 004.4
ББК 32.971.3

First published in English under the title The Art of Immutable Architecture; Theory and Practice of Data Management in Distributed Systems by Michael Perry, edition: 1.

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.



ISBN (анг.) 978-1-107-02398-7
ISBN (рус.) 978-5-93700-111-5

*Посвящается Дженни. Я бы ничего не менял*

# Оглавление

# Об авторе

Майкл Л. Перри (Michael L. Perry) опирался на работы таких математиков, как Бертран Мейер (Bertrand Meyer), Лесли Лампорт (Leslie Lamport) и Дональд Кнут (Donald Knuth), при создании математической системы для разработки программного обеспечения. Он воплотил эту систему в ряде проектов с открытым исходным кодом. Майкл часто выступает с докладами по математике и программному обеспечению на различных мероприятиях и в интернете. Вы можете узнать больше на сайте qedcode.com.

# О техническом рецензенте

Карстен Томсен (Carsten Thomsen) – в основном бэкенд-разработчик, но работает и с небольшими фронтенд-разработками. Он является автором и рецензентом ряда книг и создал многочисленные учебные курсы для компании Microsoft, связанные с разработкой программного обеспечения. Он работает как фрилансер/подрядчик в разных странах Европы, используя такие инструменты, как Azure, Visual Studio, Azure DevOps и GitHub. Является исключительным специалистом по устранению неполадок путем постановки правильных вопросов, в том числе менее логичных, переходя от наиболее логичного к наименее логичному. Ему также нравится работать с архитектурой, исследованиями, анализом, разработкой, тестированием и исправлением ошибок в программах. Он обладает хорошими навыками наставничества и руководства командой, а также исследования и представления нового материала.

# Благодарности

Как напоминает нам Толкиен, выходить за дверь – дело опасное. Первый шаг на пути, который привел к тому, что эта книга оказалась в ваших руках, был шатким. Я создавал свою первую распределенную систему и совершал все ошибки новичка. К счастью, у меня были хорошие друзья, с которыми я совершал эти ошибки.

Спасибо вам, Рассел Элледж (Russell Elledge) и Джерри Ферис (Jerry Feris), за то, что боролись вместе со мной. Вместе мы, трое друзей, изучили все неправильные способы использования TCP/IP и SOAP. Кто же знал, что трехстороннего рукопожатия недостаточно для создания нового?

Хотя первые попытки были нелегкими, мы начали разбираться. Рассел был моим постоянным соучастником, аудиторией и критиком на протяжении всего этого пути. Я должен также поблагодарить тебя за то, что ты познакомил меня с Крисом Гулдом (Chris Gould), который дал нам обоим свободу применить то, чему мы научились после той роковой первой попытки. Его поддержка позволила нам построить правильное решение на математически прочном фундаменте. Именно успех этого проекта дал мне окончательное подтверждение того, что этим понятиям *можно* научить.

Огромная благодарность Шону Уайтселлу (Sean Whitesell) за годы поддержки, ободрения и обсуждения. Ты всегда задаешь самые лучшие вопросы. И что не менее важно, умеешь мастерски объединять людей. Спасибо тебе за создание сообщества, которое помогло мне передавать идеи, вошедшие в эту книгу. И отдельное спасибо за установление окончательной связи для начала этого проекта.

Именно Шон познакомил меня с Флойдом Мэем (Floyd May). Флойд, ты такой глубокий мыслитель в области технологий, межличностных отношений и бизнеса. Ты заставил меня стать более общительным. Я не могу дождаться, когда увижу, куда тебя занесет.

Всем моим друзьям из «Улучшения»: Кори Дрю (Cori Drew), Гарольду Пулчеру (Harold Pulcher), Барри Форресту (Barry Forrest), Бену Кеннеди (Ben Kennedy), Дэвиду Вибберту (David Vibbert), Дэвиду Белчеру (David Belcher), Дэвиду О'Хара (David O'Hara)… всем Дэйвам. Мы выросли вместе. Я помню, как впервые встретил каждого из вас и все, что мы узнали с тех пор. Особое спасибо Тиму Рейберну (Tim Rayburn) за то, что помог мне вырасти как оратору, импровизатору и лидеру. И пожалуйста, не говорите Девлину Лайлзу (Devlin Liles), что я считаю его самым блестящим человеком в компании. Думаю, я бы не выдержал, если бы он узнал.

Особая благодарность Джоан Мюррей (Joan Murray) из Apress за веру в этот проект и Джилл Бальзано (Jill Balzano) за поддержку моего первого издательского опыта. И спасибо Карстену Томсену (Carsten Thomsen) за отзыв, полный улучшений и ободрения. Вы все сделали этот процесс самым увлекательным из всего, что я делал, выполняя самую сложную работу.

И наконец, моя самая искренняя благодарность моей семье. Папа, ты вдохновил меня на создание программного обеспечения. Ты обеспечил меня не только Apple II и IBM, на которых я проучился до конца школы, но и познакомил меня с первым человеком, зарабатывающим на жизнь тем, что я люблю. Ты продолжал присылать журналы Nibble и Byte, чтобы утолить мою жажду и, в конце концов, чтобы вдохновить меня писать о том, чему я научился. Я такой, какой я есть, благодаря тебе.

Дженни. Ты всегда верила в меня. Ты мой партнер и моя причина.

И Каэле. Ты заставляешь меня гордиться тобой. Я так счастлив, что мы закончили этот проект вместе.

А дорога ведет все дальше и дальше.

# Введение

Шел 2001 год. Я присоединился к команде, использующей J2EE[1] версии 1.3 для создания распределенного процессора подарочных карт. Система точек продаж была написана на Microsoft Visual C++ 6.0. Мы только узнали об этой новой вещи под названием SOAP (Simple Object Access Protocol) – простой протокол доступа к объектам. Шутка заключалась в том, что он был слишком плохо определен, чтобы называться протоколом, он не был связан с доступом к объектам и был совсем не простым. Но он давал некоторые надежды на то, чтобы заставить клиента C++ обращаться к серверу Java.

Мы добавили в свои библиотеки три новые книги. Первая была посвящена реализации клиента SOAP на C++, вторая – JAXP, Java API для обработки XML, а третья подробно описывала работу и ограничения TCP/IP. Вооружившись этими инструментами, мы начали.

Сначала задача заключалась в том, чтобы заставить две платформы общаться друг с другом. Когда мы наконец остановились на подмножестве SOAP, с которым могли работать обе стороны, мы подумали, что преодолели этот бугор. Мы не знали, что по другую сторону были горы.

Возникли проблемы с надежностью сети. Мы создали лабораторию, которая постоянно проводила транзакции каждую ночь. Утром мы проверяли баланс карт и обнаруживали, что на некоторых машинах была неверная сумма. Это приводило к целому дню копания в журналах, настраивали следующий тест, а затем запускали его до утра.

Со временем мы разработали протокол обмена сообщениями (поверх SOAP), основанный на подтверждениях и квитанциях. Одна сторона отправляла сообщение. На следующее утро мы обнаруживали пропажу сообщений. Затем получатель подтверждал, что сообщение пришло. На следующее утро мы обнаруживали дубликаты. И тогда отправитель регистрировал подтверждение. Пропавших сообщений стало меньше, но все еще не было идеально.

Потребовалось много неудачных релизов и много лет работы, в том числе по праздникам, чтобы решить все проблемы. Мы узнали о «задаче двух генералов» (TGP) и поняли, почему наш протокол обмена сообщениями был несовершенен. Затем мы узнали о конечной согласованности и получили рабочее решение. Это решение требовало, чтобы существовала некоторая неопределенность в отношении того, сколько денег осталось на подарочной карте. Мы попытались провести разговор об этом с владельцем продукта. Банкиры получают конечную последовательность денег. Владелец продукта не был банкиром.

Уроки, которые мы извлекли при работе с подарочными картами, были усвоены тяжелым трудом. «Гарантированная доставка» означает не то, что вы

---

[1] Java 2 Enterprise Edition или J2EE – ранняя версия Jakarta EE – набора спецификаций и документации для языка Java, описывающей архитектуру серверной платформы. https://ru.wikipedia.org/wiki/Jakarta_EE.

думаете. Вам нужно сначала переместить данные, а затем обработать их. Удаленные вызовы процедур (RPC) – это не вызовы процедур. В системе «клиент–сервер» нет ни одной строки кода, до которой транзакция отменяется и после которой она фиксируется. Я бы не хотел получать эти уроки снова и снова.

И вот я начал собирать эти уроки воедино и определять систему, которую я назвал историческим моделированием. Она была основана на идее, что исторические факты не могут быть изменены или уничтожены. Она опиралась на отношения предшественников и преемников между фактами. И идентифицировала факты, основываясь только на их содержании, а не местоположении. Я заполнил ноутбук примерами исторических моделей. В конце концов, я интуитивно почувствовал, какие виды решений могут быть смоделированы исторически, а какие – нет. Тогда я понял, что должен поделиться этим. Надеюсь, я смогу избавить кого-то еще от необходимости усваивать эти уроки трудным путем.

С тех пор у меня было бесчисленное множество разговоров о неизменяемых архитектурах. Я разбил тему на легко усваиваемые фрагменты для выступлений на конференциях и в группах пользователей, а также создал два фреймворка с открытым исходным кодом – Correspondence и Jinaga. Однако по отдельности они не дали возможности другим начать применять неизменяемость на практике. Принятие только части идей оставляет пробелы, которые могут быть заполнены лишь с помощью остальной части системы.

Все это привело к книге, которую вы сейчас держите в руках. Это полное описание системы, шаблонов и техник. Она предвосхищает проблемы, которые создает историческое моделирование, и предлагает решения, позволяющие обеспечить его целостную реализацию. Самое главное, в ней представлена математическая основа, которая заставляет эту технику работать.

Если вы дочитали введение до конца, то наверняка сталкивались с некоторыми из этих проблем. Возможно, вы даже нашли похожие решения. Остается только еще несколько вопросов, которые, вероятно, у вас есть по поводу этой книги. Кто должен ее читать? Что я получу от нее? Как она организована? И как мне ее читать?

Рад, что вы спросили.

## Кому следует ее прочитать

Эта книга предназначена в первую очередь для трех аудиторий: лиц, принимающих решения, конструкторов систем и создателей инструментов. Вы – лицо, принимающее решения, если вы определяете проблемы, для которых хотите создать решения. Ваша должность может быть технический директор, владелец продукта или аналитик бизнес-систем. Существуют проблемы, которые вы можете передать на аутсорсинг, есть проблемы, для которых вы можете купить решения, а есть проблемы, которые определяют основную ценность вашего бизнеса. Вам необходимо найти подходящих людей для решения проблем третьего типа. Чтобы найти их, вам нужно уметь с ними разговаривать. А после того, как вы пригласите их на работу, нужно понять, чем они занимаются. Если ваша основная бизнес-проблема похожа на ту, которую можно решить с помощью неизменяемой архитектуры, эта книга поможет вам создать такую команду и провести эти беседы.

Или, возможно, вы занимаетесь созданием систем. Вы являетесь членом команды, привлеченной для создания ценностей в основной области бизнеса. Ваша должность может быть разработчик, инженер по контролю качества или дизайнер пользовательского интерфейса. Вы знаете, как решать проблемы. Но было бы здорово иметь несколько готовых решений для наиболее распространенных проблем распределенных вычислений. Вы хотите знать, что все нестандартные ситуации учтены. Вы хотите иметь общий язык для обсуждения решений с людьми, которые помогают вам находить их. Если ваши задачи по разработке программного обеспечения требуют создания согласованных распределенных систем, то эта книга даст вам такие инструменты.

Наконец, вы, как и я, возможно, являетесь создателем инструментов. Вы – множитель силы. Ваши разработки дают возможность другим создавать решения быстрее, более предсказуемо и более эффективно. Вы можете быть архитектором решений или сопровождающим открытого исходного кода. Если у вас есть команда, вы хотите, чтобы ее члены были сосредоточены на создании бизнес-ценностей, а вы позаботитесь о «сантехнике». Если вы обслуживаете сообщество, вы хотите, чтобы потребители могли быстро изучить и применить ваш фреймворк для создания надежных систем. В любом случае, в этой книге изложены математика, алгоритмы и шаблоны, которые гарантируют корректность ваших решений.

## Что вы получите от этого

У меня есть секрет. Это книга по математике. Не говорите об этом никому, кто не дочитал до конца введение.

Математика – величайшее изобретение человечества. Она удивительна в своей способности описывать мир природы. Она поразительно применима к широкому кругу проблем. И это единственный способ, с помощью которого мы можем быть уверены в чем-либо.

Обычно мы убеждаемся в том, что поняли что-то правильно, проверяя это. Мы применяем наше решение в одной ситуации и смотрим, получим ли мы ожидаемый результат. Затем пробуем другой сценарий и смотрим, что получится. Если мы действительно хороши, то сможем представить несколько непредвиденных условий и протестировать их. Но непредвиденное очень трудно предугадать.

Тестирование – это сбор опытных данных. Оно дает уверенность в том, что система ведет себя так, как ожидается, в определенных случаях, но не дает никакой уверенности в том, что вы ничего не упустили.

Знание требует математических выводов. Если что-то доказано математически, то вы можете быть уверены, что это будет истинно, независимо от того, какой тест вы попробуете провести. Теорема Пифагора верна для любого прямоугольного треугольника. Геометрия Евклида верна для всех фигур на плоскости. Если ваши рассуждения безупречны, вы можете быть уверены, что не пропустили ни одного крайнего случая.

Это не означает, что математические истины универсальны. Дело в том, что они имеют известные ограничения. Деление работает только для ненулевых делителей. Теорема Пифагора действует лишь на плоскости. Правила дедукции

говорят нам, как перенести эти ограничения на решение, чтобы точно знать, где это решение применимо, а где нет.

Эта книга применяет математическую строгость к проблеме распределенных вычислений. Она не является первой в данной области, но предлагает полное и практическое решение. Если вы будете следовать дедуктивным рассуждениям над проблемой и внесете ограничения распределенных систем в свои вычисления, то в итоге вы получите понимание границ решения. Эта книга – ваш путеводитель в этом процессе.

## Как она организована

Книга условно разделена на три части, предназначенные для трех основных аудиторий. Лицам, принимающим решения, необходимо прочитать только первую часть, которая включает в себя первые три главы. В этой части вы сначала узнаете, почему неизменяемость так важна. Затем исследуете пространство альтернатив и познакомитесь с историческим моделированием. Наконец, вы узнаете, как читать историческую модель, чтобы более эффективно общаться со своей командой. Вы можете прекратить чтение, когда мы перейдем к глубокой математике.

Специалисты по созданию систем захотят прочесть вторую часть. Она включает в себя главы с четвертой по восьмую. Мы глубоко погружаемся в математические основы неизменяемости, причинности и бесконфликтных реплицируемых типов данных (conflict-free replicated data types – CRDT). Затем увидим, как применять эти математические рассуждения для анализа систем, построения машин состояний и соблюдения правил безопасности. Именно эти инструменты понадобятся вам для создания надежных распределенных систем. В завершение данного раздела приведен каталог основных шаблонов, которые помогут вам начать строить исторические модели.

Люди моей профессии, создатели инструментов захотят дочитать до конца. В третьей части мы разберем компоненты компьютерной системы и узнаем, как использовать их в неизменяемой архитектуре. Мы научимся обновлять пользовательский интерфейс с помощью инверторов запросов. Мы будем хранить неизменяемые записи в реляционной базе данных, будем надежно и безопасно обмениваться неизменяемыми сообщениями в сетях различных типов. В конце концов, мы собираем все вместе и описываем экосистему, состоящую из совместных приложений, генерирующих новое поведение на основе общих спецификаций. Это нечто поистине прекрасное и вдохновляющее, и я надеюсь, что вы последуете за мной до конца.

## Как ее читать

Теперь, когда вы знаете, что это книга по математике, у вас могут возникнуть некоторые сомнения по поводу того, как вы будете ее читать. Возможно, вы с трудом проходили алгебру или забросили вычисления. Вы можете подумать, что математика не для вас.

Я считаю, что математика подходит всем. И моя цель в данной книге – доказать это. Математика – это не что иное, как применение логических рас-

суждений к символическим представлениям абстрактных понятий. С другой стороны, программирование – это применение логических операций к символическому языку, описывающему общие правила. Иными словами, это одно и то же. Если вы программист, то вы – прикладной математик.

Одна из проблем математики – это жаргон. Для того чтобы эффективно общаться друг с другом, математикам приходится придумывать слова для обозначения идей. К сожалению, естественный язык ограничен, и все хорошие слова уже заняты. И поэтому математики либо придумывают новые слова, либо используют термины, которые означают почти то, что нужно. В этой книге мы будем говорить, например, о свойствах полурешетки связности. Но я постараюсь не использовать эти слова, если смогу избежать этого. А если избежать не удастся, я буду давать им четкое определение.

Еще одна проблема с математикой – это то, как она написана. Математические работы имеют предсказуемую форму. Они начинаются с аннотации, затем полностью определяют проблему. Далее следуют раздел за разделом лемм и предложений, выстраивающих аргументацию. Каждое утверждение обосновывается предыдущими утверждениями. Наконец, как у М. Найт Шьямалана (M. Night Shyamalan), следует поворот сюжета, последнее утверждение рассматривает всю аргументацию в перспективе, и появляется результат.

Хотя мне очень нравятся хорошие математические статьи, я не читаю их так, как они написаны. Я бегло просматриваю первые несколько абзацев в поисках мотивации проблемы. Сканирую заголовки, чтобы понять суть аргументов. Я хочу знать, почему каждое утверждение доказано и каков будет его вклад в общее дело. Я хочу знать, как будет развиваться история, прежде чем потратить время на ее понимание.

Я написал эту книгу так, как я читаю статьи по математике. В каждом разделе вы поймете, чем обусловлен тот или иной результат. Затем вы увидите набросок основных рассуждений. Не будет загадкой, почему каждый шаг находится на отведенном ему месте. Потом каждый из этих шагов будет обоснован с той строгостью, которой он требует.

Я ожидаю, что это повлияет на то, как вы будете читать книгу. Если вам нужны только результаты, вы можете прочитать всего один-два абзаца после заголовка раздела. Если вы хотите узнать, почему или как, то продолжите читать дальше, чтобы понять аргументацию. А если вам нужно убедиться, то дочитайте весь раздел до конца. Важно, что вы можете прекратить чтение, когда оно становится слишком глубоким, и перейти к следующему разделу. Вы не пропустите ничего важного для себя.

Если вы прочитали этот раздел, ничего не пропустив, то я искренне рад, что вы у меня есть. Вы – один из моих людей. С вашей помощью мы сможем создать программное обеспечение, которое необходимо миру. Мы сделаем его надежным, эффективным и правильным. И это даст нашим пользователям автономию, необходимую для того, чтобы они могли выполнять свою работу творчески и уверенно, зная, что мы обеспечили математическую строгость.

# ЧАСТЬ I

# Определение

# Глава 1

# Почему неизменяемая архитектура

Распределенные системы – это сложно.

Большинство из нас пользовались веб-сайтом для покупки товара. Возможно, вы видели страницу покупки, на которой есть предупреждение «**Не нажимайте кнопку "Отправить" дважды!**». Может быть, вы пользовались сайтом, который просто отключает кнопку покупки, после того как вы ее нажмете. Авторы этого сайта столкнулись с одной из сложных проблем распределенных систем и не знали, как ее решить. Они переложили ответственность за предотвращение дублирования расходов на потребителя.

Возможно, вы пользовались мобильным приложением в поезде. Поезд въезжает в туннель как раз в тот момент, когда вы сохраняете некоторые данные. Мобильное приложение работает несколько секунд, прежде чем вы это поймете. Выедет ли поезд из туннеля до того, как приложение прекратит попытки сохранения данных? Исправит ли приложение ошибку, когда связь будет восстановлена? Или вы потеряете свои данные и вам придется вводить их снова?

Если вы участвуете в создании распределенных систем, от вас ожидают, что вы будете находить, исправлять и предотвращать подобные ошибки. Если вы работаете в отделе контроля качества, ваша работа заключается в том, чтобы представить все возможные сценарии, а затем воспроизводить их в лаборатории. Если вы разработчик, вам нужно написать код для всех различных исключений и условий такого процесса. А если вы занимаетесь архитектурой, вы отвечаете за разрубание гордиева узла возможных сбоев и смягчение последствий. Это тонкий процесс, с помощью которого мы создаем системы, управляющие нашим обществом.

## Решение проблемы неизменяемости

Распределенные системы трудно писать, тестировать и поддерживать. Они ненадежны, непредсказуемы и небезопасны. Процесс, с помощью которого мы их создаем, обязательно упустит из виду дефекты, которые негативно повлияют на наших пользователей. Но это не ваша вина. До тех пор, пока мы будем полагаться на отдельных людей в поиске, устранении и смягчении этих проблем, дефекты будут упущены.

В этой книге рассматривается другой процесс создания распределенных систем. Вместо того чтобы связывать программы между собой и тестировать дефекты, этот подход начинается с фундаментального представления бизнес-проблемы, которая *объединяет* машины. И это фундаментальное представление является *неизменным*.

На первый взгляд, неизменяемость – это простая концепция. Запишите некоторые данные и убедитесь, что они никогда не изменятся. Они не могут быть изменены, обновлены или удалены. Они неизменны. Неизменяемость решает проблему распределенных систем по одной простой причине: каждая копия неизменяемого объекта так же хороша, как и любая другая копия. До тех пор, пока объекты никогда не меняются, синхронизация удаленных копий является тривиальной задачей.

## Проблемы с неизменяемостью

К сожалению, неизменяемость противоречит тому, как на самом деле работают компьютеры. Машина имеет ограниченный объем памяти. Машины работают, изменяя содержимое ячеек памяти с течением времени, чтобы обновить свое внутреннее состояние. Поэтому первая проблема моделирования неизменяемых данных на компьютере заключается в том, как представить их в изменяемой памяти.

Вторая проблема заключается в том, что когда мы смотрим на мир проблем, которые хотим решить, мы видим изменения. Люди меняют свои имена, адреса и номера телефонов. Остатки на банковских счетах растут и падают. Имущество переходит из рук в руки, и право собственности переходит. Как же нам смоделировать изменяющееся проблемное пространство с неизменными данными?

Наш первоначальный инстинкт заключается в том, чтобы моделировать изменчивый мир в изменчивом пространстве компьютера. Именно это решение привело нас к созданию программ и баз данных на основе изменений. Программы содержат операторы присваивания, базы данных содержат операторы ОБНОВИТЬ. Когда мы соединяем эти программы и базы данных вместе для создания распределенных систем, возникает безумное непредсказуемое поведение. И перед нами встает бесконечная задача тестирования, пока все эти аномалии не исчезнут.

## Начинаем новое путешествие

Цель этой книги состоит в том, чтобы вместо этого смоделировать бизнес-домен как одну большую неизменяемую структуру данных. Для одной машины или базы данных было бы невозможно разместить всю эту структуру. Да это и нежелательно. Поэтому книга также стремится продемонстрировать, как реализовать подмножества этой структуры данных в отдельных базах данных, программах и машинах. Эти компоненты взаимодействуют через хорошо продуманные протоколы, которые учитывают особенности распределенных систем, чтобы развивать неизменяемую структуру данных с течением времени.

Данное решение не является новым. На протяжении всей этой книги мы будем обращаться к исследованиям прошлого в виде статей по математике и информатике. Каждое утверждение обосновано, ни один из выводов не является оригинальным. Я надеюсь лишь собрать все эти знания воедино, чтобы начать ваше путешествие к более надежным, устойчивым и безопасным распределенным системам. Давайте начнем это путешествие с понимания проблемы распределенных вычислений.

## Ошибки распределенных вычислений

В период с 1991 по 1997 год инженеры компании Sun Microsystems собрали список ошибок, которые программисты часто допускали при написании программного обеспечения для сетевых компьютеров. Билл Джой (Bill Joy), Дэйв Лайон (Dave Lyon), Питер Дойч (Peter Deutsch) и Джеймс Гослинг (James Gosling) составили каталог из восьми предположений, которых разработчики обычно придерживались в отношении распределенных вычислений. Эти предположения, хотя и очевидно неверные, когда они высказаны прямо, тем не менее определяют многие из решений, которые инженеры Sun нашли в системах того времени.

Заблуждения таковы:

- сеть надежна;
- задержка равна нулю;
- пропускная способность бесконечна;
- сеть безопасна;
- топология не меняется;
- имеется один администратор;
- транспортные расходы равны нулю;
- сеть однородна.

Хотя с момента написания этого списка прошло много лет, многие из этих предположений продолжают быть общепринятыми. Я могу вспомнить несколько случаев, когда меня удивляло, что программа, которая безупречно работала на *локальном компьютере*, быстро выходила из строя при развертывании в тестовой среде. Программа содержала скрытые предположения о том, что сеть надежна, задержка нулевая и топология не меняется. Вот примеры только этих трех предположений.

### Сеть ненадежна

Один из способов, с помощью которого эти заблуждения проявляются в современных системах, – это когда удаленный программный интерфейс приложения (API) рассматривается так, как будто это вызов функции. Несколько сервисов платформы продвигают эту абстракцию, включая вызовы удаленных процедур, .NET Remoting, Distributed COM, SOAP и SignalR. Когда удаленный вызов выглядит как вызов локальной функции, разработчику легко забыть о том, что сеть ненадежна.

Каждый раз, когда вы вызываете функцию, вы можете быть уверены, что выполнение программы продолжится с ее первой строки. И если функция

дойдет до оператора возврата, вы можете быть уверены в том, что следующей строкой будет та, которая следует за вызовом функции. Вызовы удаленных процедур, однако, не дают таких гарантий. Они могут потерпеть неудачу при вызове или при возврате. Вызывающий код не сможет определить, где именно это произошло.

Абстракция, которая скрывает факт сетевого перехода, оказывает плохую услугу потребителям. Пытаясь сделать вещи более простыми и привычными, она делает вид, что неудобная правда может быть проигнорирована. Такие абстракции позволяют разработчикам поверить в заблуждение, что сеть каким-то образом надежна.

## Время задержки не равно нулю

Современные веб-приложения отказались от клиентских прокси в пользу более явных REST API. Эти API позволяют избежать ошибки представления удаленной машины так, как если бы она была библиотекой функций, которые можно надежно вызывать. Вместо этого они представляют мир как паутину взаимосвязанных ресурсов, каждый из которых отвечает на небольшой набор HTTP-методов. К сожалению, при таком стиле программирования легко забыть, что время задержки не равно нулю.

Некоторые из HTTP-методов гарантированно идемпотентны. Если клиент дублирует запрос, сервер обещает не дублировать эффект. Протокол не имеет возможности обеспечить соблюдение этой гарантии, но приложения на стороне сервера обычно поддерживают данный договор. Примерами методов HTTP, которые являются идемпотентными, являются PUT и PATCH. Метод HTTP, который не гарантирует идемпотентность, – это POST.

В интернете HTTP POST часто используется для отправки формы. Когда веб-приложение реагирует быстро, отсутствие гарантии идемпотентности не имеет большого значения. Но по мере увеличения задержки пользователь начинает сомневаться, действительно ли он нажал на кнопку отправки. А если эта кнопка инициировала покупку, то пользователь начинает задумываться, не будет ли с него снята двойная оплата, если он попытается сделать это снова. Конечный пользователь не имеет хороших средств защиты во время длительной задержки после нажатия кнопки «Купить». У разработчика клиентского приложения также нет хорошего ответа на задержку при отправке POST.

Не существует корректного использования API, в котором присутствуют неидемпотентные сетевые запросы. Поскольку задержка не равна нулю, всегда будет время, в течение которого клиент не уверен в том, получил ли сервер запрос. Когда задержка превышает время, которое клиент готов ждать, он должен сделать выбор: либо прервать попытку, либо повторить ее. Если клиент прерывает попытку, то он не знает, был ли запрос обработан. А если он повторяет попытку, то эффект может быть продублирован.

Метод POST действительно является частью спецификации HTTP. И эта спецификация не дает никаких гарантий относительно его идемпотентности. Но любой API, который включает неидемпотентный POST, делает неверное предположение, что задержка равна нулю. Это заставляет клиента делать неверный выбор, когда данное предположение оказывается ложным.

## Топология не меняется

Большинство систем управления базами данных включают концепцию, которая заставляет разработчиков предполагать, что топология не меняется. Эти базы данных позволяют легко установить автоинкрементируемый идентификатор (auto-incremented ID) записи. Каждый раз, когда вставляется новая запись, база данных генерирует следующий номер в последовательности. Этот номер в дальнейшем используется для идентификации записи.

Автоинкрементируемый идентификатор требует, чтобы топология оставалась постоянной на протяжении всего многоэтапного процесса. Представьте себе веб-приложение, которое вставляет данные формы пользователя в базу данных, а затем перенаправляет пользователя на страницу, представляющую эти новые данные. Чтобы выполнить это с помощью автоинкрементируемого идентификатора, браузер должен ждать, пока запрос пройдет весь путь до базы данных и пока ответ вернется обратно, прежде чем он сможет узнать URL следующей страницы. Приложение предполагает, что топология за это время не изменится.

На первый взгляд это может показаться верным предположением. Обычно это так и есть. Изменения топологии сервера происходят редко, а сетевые запросы обычно выполняются быстро (задержка равна нулю). Однако для веб-приложения с высокой посещаемостью никогда не будет момента, в течение которого не обрабатывается ни один запрос. Предположение о том, что топология не меняется, будет нарушено для некоторых запросов.

Топология может измениться во время обновления системы. Она обязательно изменится во время аварийного переключения. И она снова изменится при возврате назад после устранения аварии. При изменении топологии база данных, к которой поступает запрос, будет не той, в которой была создана исходная страница. Вместо этого база данных будет репликой оригинала. Если реплика лишь немного отстает от оригинала, то изменение топологии будет заметным. И оно будет заметным, потому что, опять же, задержка не равна нулю.

Использование автоинкрементных идентификаторов повсеместно. Они являются выбором по умолчанию для большинства баз данных. И все же их использование противоречит предположению о том, что топология не будет меняться.

## Изменение предположений

Заблуждения распределенных вычислений – это простые предположения. Мы делаем их, потому что наши инструменты, спецификации и обучение привели нас к этому. Неидемпотентный метод POST является валидной частью спецификации HTTP. Автоинкрементные идентификаторы являются полезной особенностью большинства систем управления базами данных. Почти каждый учебник по разработке приложений научит новичка использовать эти возможности. Тот факт, что при этом он делает неверное предположение, даже не приходит ему в голову.

Инструменты, которые мы используем, и шаблоны, которым мы следуем сегодня, – все это появилось в те времена, когда предположения о высокой надежности, нулевой задержке и неизменности топологии не были заблуждениями.

Внутрипроцессные вызовы процедур абсолютно надежны. Последовательные операторы программы имеют очень малые, очень предсказуемые характеристики задержки. А последовательные счетчики в цикле for никогда не вернутся в начало функции, чтобы обнаружить, что топология кода изменилась. Именно когда мы применяем эти абстракции при вызове удаленных процедур (Remote Procedure Call – RPC), в сетевых запросах и автоинкрементируемых идентификаторах, возникают проблемы. Когда мы применяем языки и шаблоны прошлого к проблемам современных распределенных систем, неудивительно, что программисты делают неверные предположения.

Все заблуждения при распределенных вычислениях проистекают из-за одного простого допущения – распределенные системы создаются с использованием инструментов, предназначенных для работы в одном потоке на одном компьютере. Разработчики представляют себе быструю, изолированную, неизменную, последовательную среду выполнения, а затем рассматривают идиосинкразии распределенных систем как частные случаи. Дублирование транзакции из-за тайм-аута в сети не является ошибкой. Столкновение идентификаторов, вызванное отказом базы данных, не является дефектом. Это реалии распределенных систем, которые мы не можем обойти кодом или протестировать. Они требуют нового набора инструментов, моделей и предположений.

## Неизменяемость меняет все

В 2015 году Пэт Хелланд (Pat Helland) написал статью «Неизменяемость меняет все»[1], в которой анализируется несколько вычислительных решений, основанных на неизменяемости. В ней показано, что неизменяемость решает многие проблемы на нескольких уровнях вычислительной абстракции. На одном конце спектра низкоуровневые системы хранения данных используют семантику копирования при записи для защиты от износа носителя. На другом конце приложения накапливают факты, доступные только для чтения, и получают текущее состояние. Данная статья не претендует на новые идеи, а лишь указывает на то, что во всех этих решениях присутствует общая нить – неизменяемость.

В прошлом компьютеры были медленными, дорогими и ограниченными машинами, которые могли оперировать только с небольшими наборами данных. Сегодня это быстрые, дешевые и способные рабочие лошадки, которые хранят огромное количество данных. Если раньше разработчикам приложений приходилось оптимизировать хранение данных, перезаписывая информацию, когда она больше не нужна, то сегодня мы можем позволить себе сохранять все. Нет экономической необходимости обновлять или уничтожать информацию.

В то же время современные компьютеры гораздо более связаны между собой, чем это было в прошлом. Вместо того чтобы размещать рабочую нагрузку вместе с данными, на которых она работает, мы перешли в мир микросерви-

[1] *Helland Pat.* (2015). Immutability Changes Everything // http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.

сов и мобильных устройств, которые широко обмениваются данными. Многие машины разделяют вычислительное бремя и бремя хранения данных, которое раньше выполнялось одной. В результате координация стала более дорогой, несмотря на то что вычисления стали дешевыми.

И если в прошлом хранить неизменяемые копии данных было дорого, нынешние архитектурные ограничения *требуют* этого. Данные не только дешевле, чем раньше, но создание неизменяемых копий фактически *обеспечивает* создание решений, которые масштабируются на несколько машин. Когда две машины совместно используют изменяемые данные, им необходимо координировать свои действия при изменении этих данных. Им может понадобиться блокировать друг друга, чтобы гарантировать, что только одна из них может изменять данные в любой момент времени. Но когда эти данные не могут меняться, координация или блокировка не требуется. Снижение затрат обеспечивает неизменяемость, а неизменяемость обеспечивает современную архитектуру.

## Совместное изменяемое состояние

Многие трудные проблемы в вычислениях – это проблемы, которые мы создали для себя сами. Возьмем, к примеру, проблему общего изменяемого состояния в многопоточной системе. Один поток записывает исходные данные в общую область памяти, а другой поток выполняет в ней вычисления. Эти два потока должны быть тщательно скоординированы, чтобы один из них не записывал данные в общую память до того, как другой закончит чтение из нее. Если первый поток перезапишет данные, пока второй продолжает вычисления, результаты будут полной бессмыслицей. Обычно мы решаем подобную проблему с помощью блокировки, ограничивающей возможность масштабирования.

Но есть решение, которое не ухудшает масштабируемость. Вместо блокировки мы можем использовать неизменяемые структуры данных. Вместо того чтобы перезаписывать память следующим набором данных, первый поток просто выделит новую память. Когда завершается построение структуры данных, первый поток передает указатель второму. С этого момента ни один поток не может изменить содержимое этой области памяти. Она остается полностью неизменяемой.

На первый взгляд кажется, что мы улучшили масштабируемость за счет снижения эффективности памяти. Кажется, что, вместо того чтобы изменять только одну небольшую часть структуры данных, мы должны создавать целую копию при каждой операции. Если бы это было так, было бы трудно оправдать этот компромисс, даже с учетом снижения стоимости хранения данных. К счастью, нам не приходится идти на такой компромисс.

## Структурное разделение

Тот факт, что структуры данных должны быть неизменяемыми, открывает новые возможности. По мере того как мы создаем новые структуры данных, мы можем повторно использовать существующие части старых структур данных. При этом нет необходимости копировать эти части, потому что мы уже установили, что они не будут меняться. Мы просто создаем новые элементы данных

для представления тех, которые «изменились», и разрешаем им указывать на те, которые не изменились.

Такая техника называется *структурным разделением*. Это обычная оптимизация, которая *обеспечивается* неизменяемыми структурами данных. Возьмем, например, двоичное дерево, показанное на рис. 1.1. Каждый узел дерева содержит часть данных, в нашем случае число. Он также содержит два указателя, один на число, которое меньше, чем этот узел, и один на число, которое больше. Нахождение конкретного числа в этой структуре данных является быстрым, потому что вы идете вниз, спрашивая «меньше чем или больше чем» на каждой остановке.

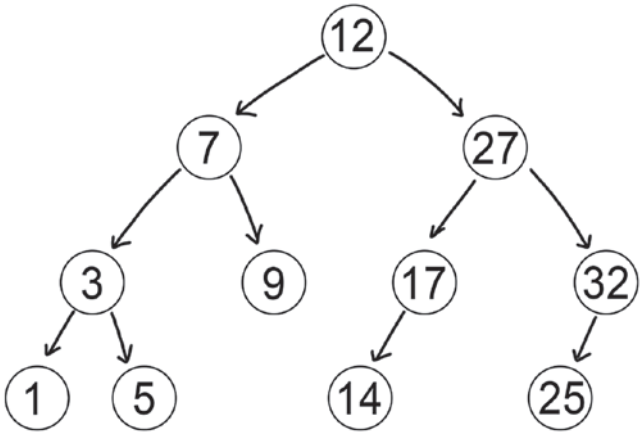


**Рис. 1.1.** Двоичное дерево чисел

Чтобы вставить новое число в двоичное дерево, сначала нужно найти место для него. Пройдя до места, где оно должно находиться, вы обнаружите, что оно либо меньше числа, у которого нет левого пути, либо больше числа, у которого нет правого пути. Оказавшись там, вы захотите «изменить» этот узел, чтобы добавить новый путь. Однако изменение узла не разрешается – все они являются частью неизменяемой структуры данных. Поэтому вместо этого вы создаете новый узел.

Этот новый узел должен быть левее или правее родительского, и поэтому вы хотите «изменить» и этот узел. Но опять же, изменение родителя не допускается. Поэтому вы создаете нового родителя, который указывает на новый дочерний узел.

Продолжая подниматься вверх по дереву, вы в конце концов достигнете корня, как показано на рис. 1.2. Независимо от того, куда вы вставляете новое число, вы всегда в итоге создаете новый корневой узел. Этот узел фактически является новой версией дерева. Он представляет собой форму дерева после вставки. Предыдущий корневой узел все еще существует, и узлы, на которые он указывает, не были изменены. Любые потоки, выполняющие параллельный поиск в этой версии дерева, могут спокойно продолжать это делать. Они не будут затронуты новым деревом, которое разделяет большую часть своей структуры со старым.

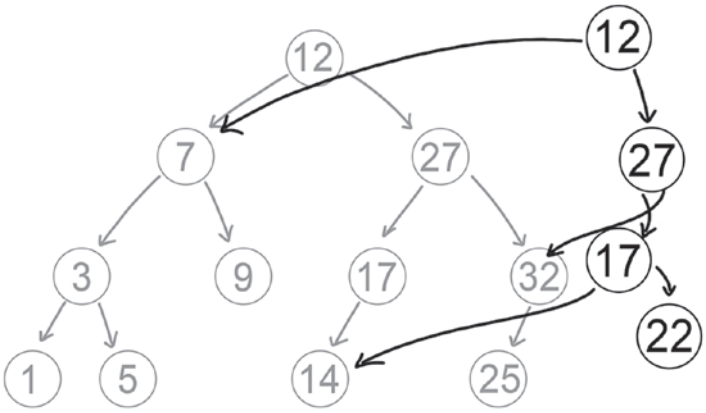


**Рис. 1.2.** После вставки числа 22 новая версия двоичного дерева разделяет большую часть своей структуры с предыдущей версией

Эта оптимизация была бы невозможна, если бы потоки могли изменять эти структуры данных. Благодаря совместному использованию структуры эти две версии дерева становятся чувствительными к модификации. Только потому, что мы договорились *не* изменять узлы, мы можем избежать глубокого разделения структуры. Неизменяемость обеспечивает структурное разделение, а структурное разделение оптимизирует неизменяемость.

## Проблема двух генералов

Нигде в вычислительной технике неизменяемость не является более ценной, чем при обмене данными между машинами. Но прежде чем мы сможем понять причину этого, мы должны сначала понять масштаб проблемы. И нет лучшего способа сделать это, чем притча о двух генералах[1].

Представьте себе осажденный город. В его стенах оборона непреодолима. Прямая атака почти наверняка потерпит неудачу. За пределами города находятся две армии, которым удалось отрезать его от линий снабжения. Генералы этих армий ждут, наблюдая, как город медленно слабеет под блокадой.

В какой-то момент оборона города станет достаточно слабой для атаки. Генералы этих двух армий (одной – на востоке, второй – на западе) постоянно наблюдают за ситуацией через сеть разведчиков, шпионов и гонцов. Каждый день они определяют, достаточно ли слаб город. Когда приходит время, они готовят атаку на следующий день. Эта ситуация показана на рис. 1.3.

[1] *Akkoyunlu E. A., Ekanadham K., Huber R. V.* Some constraints and tradeoffs in the design of network communications. ACM SIGOPS Operating Systems Review. November 1975.

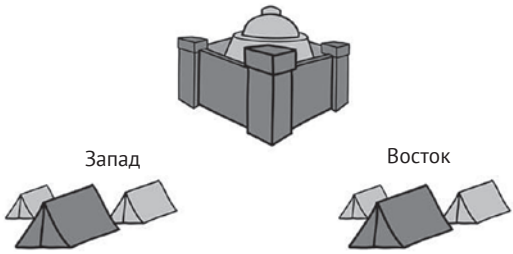


**Рис. 1.3.** Две армии расположились лагерем у осажденного города

Атаки одной армии недостаточно. Атака была бы отбита и атакующая армия уничтожена. Оставшаяся армия была бы не в состоянии поддерживать блокаду, и поэтому вскоре она была бы разгромлена. Только скоординированная атака с востока и запада позволит завоевать город.

Теперь представьте, что вы – генерал армии запада. Ваш партнер на востоке отделен от вас вражеской территорией. Вы не можете общаться напрямую, а можете только послать гонцов через враждебную местность без гарантии успеха. Любое сообщение может быть потеряно, его носитель убит или захвачен в плен. Вы двое должны разработать метод надежной связи, построенной из ненадежных компонентов.

Если вы на западе решите, что город достаточно слаб и что время для атаки наступило, вы начнете готовить свою армию. Вы также пошлете гонца на восток, чтобы сообщить другому генералу, что вы нападете утром. Если гонец прибудет благополучно, то восточный генерал сможет начать подготовку и присоединиться к вам в атаке. Благодаря вашим совместным усилиям атака, скорее всего, будет успешной.

Но если гонец будет убит или захвачен, сообщение не дойдет. Если это произойдет, утром ваша армия отправится в одиночную атаку на город. Ваша армия будет уничтожена, и осада будет проиграна. Как показано на рис. 1.4, вы не знаете, как действовать дальше. Поэтому до наступления утра вы должны быть уверены в том, что послание получено.


**Рис. 1.4.** Западный генерал не знает, смогут ли они атаковать

## Заранее подготовленный протокол

Давайте попробуем разработать протокол, который даст нам некоторую уверенность в том, что сообщение было получено. Предположим, вы просите восточного генерала отправить в ответ гонца с подтверждением того, что ваше сообщение было получено. Теперь если вы получите подтверждение до утра, то сможете уверенно начать атаку. Вы знаете, что восточный генерал получил сообщение и присоединится к вам на поле боя. Если же, с другой стороны, вы не получите подтверждения, то отмените атаку, не зная, дошел ли первоначальный гонец. Как генерал западной армии вы можете быть уверены, что не атакуете, пока не узнаете, что восточный генерал получил ваше сообщение.

Но хотя этот протокол дает такие гарантии генералу запада, он не делает этого для восточного генерала. Представьте себе, что вы находитесь на востоке и получили сообщение о том, что западная армия будет атаковать утром. У вас есть достаточно времени, чтобы начать подготовку своей армии. И, согласно протоколу, вы отвечаете подтверждением. Если сообщение с подтверждением дойдет до генерала запада, то атака пройдет по плану.

Но если это сообщение будет утеряно, то западный генерал не будет атаковать. Помните, он ждет подтверждения, чтобы знать, что вы получили его сообщение. Если вы атакуете утром, не зная, что западный генерал получил ваше подтверждение, то ваша армия может быть разбита. И тогда вы окажетесь в неопределенности, как на рис. 1.5.

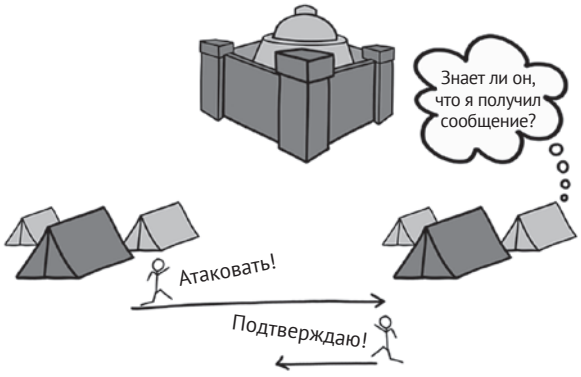


**Рис. 1.5.** Восточный генерал не уверен

## Уменьшение неопределенности

Этот протокол недостаточен. Вы пробуете различные стратегии, чтобы улучшить его. Первая стратегия состоит в том, чтобы просто послать больше гонцов. Вместо того чтобы полагаться на одного гонца, вы посылаете двух. Вероятность того, что два сообщения будут потеряны, конечно, меньше, чем вероятность потери одного. Но эта вероятность не равна нулю. И вот вы пробуете снова.

Вы можете послать трех, четырех гонцов. Выбирайте любое число, какое пожелаете. По мере того как вы увеличиваете число, вероятность потери сообщения становится все ближе и ближе к нулю. Но она никогда не достигает его.

Вы никогда не сможете выбрать число гонцов достаточно большое, чтобы быть уверенным, что сообщение будет получено.

И тогда вы меняете свой подход. Вы посылаете гонцов с постоянной скоростью, пока не будет получен ответ. С запада, когда вы решаете атаковать, вы посылаете гонцов с сообщением об *атаке* раз в десять минут. Когда вы получаете первое *подтверждение* с востока, вы прекращаете отправку сообщений. Что касается генерала на востоке, он будет отвечать *подтверждением* каждый раз, когда сообщение об *атаке* будет получено. Пока он будет получать постоянный поток сообщений об *атаке*, он будет отвечать на них с той же скоростью подтверждением. И как только этот поток прекратится, он может считать, что подтверждение получено.

А это правильно? Можно ли воспринимать отсутствие сообщений как сигнал? Возможно ли, что шесть гонцов в час продолжают отправляться с запада, но все они захвачены? Генерал на востоке не может этого исключить. И поэтому он все еще рискует атаковать утром без поддержки с запада.

## Дополнительное сообщение

Поэтому, как восточный генерал, вы выдвигаете дополнительное требование к протоколу. В дополнение к сообщению об атаке с запада и подтверждению с востока вы требуете, чтобы запад ответил подтверждением. Если вы на востоке получите подтверждение до утра, то вы знаете, что подтверждение было получено на западе. Поэтому вы можете атаковать с уверенностью, зная, что западный генерал получил подтверждение и поэтому присоединится к вам. Но если вы не получили подтверждения, тогда должны воздержаться.

Хотя это новое сообщение дает новые гарантии генералу востока, оно снова запутывает ситуацию на западе. Когда западный генерал посылает подтверждение, он не может узнать, было ли оно получено. Если да, то восточный генерал атакует. Если нет, то он воздержится. И как показано на рис. 1.6, у него нет никакой уверенности в том, что его атака будет поддержана.

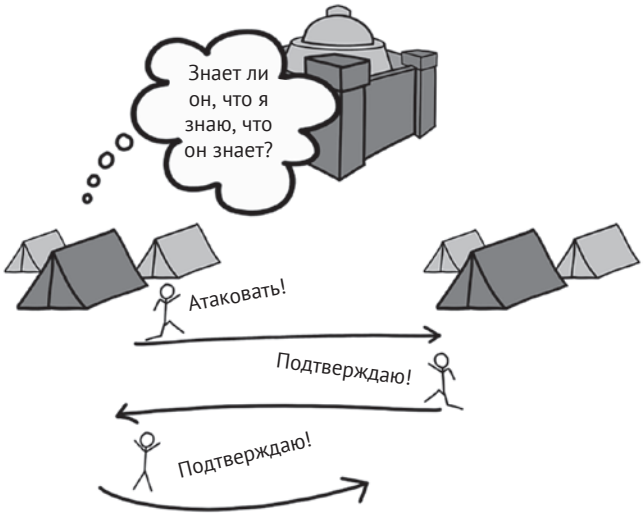


**Рис. 1.6.** Западный генерал снова не уверен, что восток будет атаковать

Добавление еще одного сообщения только перенесло неопределенность на другую сторону. На самом деле это не решило проблему. Мы все еще не нашли протокол, который обеспечит, что обе армии либо атакуют, либо воздержатся, если два генерала могут общаться только посредством ненадежных сообщений.

И действительно, мы никогда его не найдем.

## Доказательство невозможности

Проблема двух генералов, как назвал ее Джим Грей (Jim Gray) в 1978 году[1], не имеет решения. Не существует конечного протокола, который может дать обоим генералам взаимную уверенность в достижении соглашения. Я не просто говорю, что никто не нашел решения. Я говорю, что решения не может существовать.

Е. А. Аккоюнлу (A. Akkoyunlu), который опубликовал оригинальную проблему и доказательство невозможности в 1975 году[2], назвал эту взаимную гарантию *полным статусом*. Он описал межпроцессные коммуникационные протоколы, которые согласовывают транзакции между участниками. Протокол в идеале должен обеспечивать этот статус для участников относительно результата каждой транзакции. Аккоюнлу доказал, что распределенная система не может достичь полного статуса за конечное число сообщений.

Его доказательство не требует, чтобы мы исчерпали все возможные решения. Оно не оставляет места для хитроумных уловок, о которых мы не подумали. Вместо этого оно основано на противоречии. Пусть кто угодно придумает протокол и принесет его Аккоюнлу, утверждая, что он обеспечивает полный статус. Даже не зная, как работает этот протокол, он показывает, что он не подтверждает данное утверждение.

Предположим, что вы представляете протокол, который, как вы утверждаете, обеспечивает полный статус двум генералам после обмена конечным числом сообщений. В конце этого обмена оба генерала будут знать, что другой собирается атаковать. Если генералы следуют этому протоколу и ни одно сообщение не было потеряно, то существует минимальное количество сообщений, чтобы достичь этого состояния. Назовем это число N. Число N является специфическим для протокола.

Поскольку N – это наименьшее количество сообщений, которыми необходимо обменяться для достижения полного состояния, мы знаем, что меньшее число будет недостаточным. В частности, мы не достигли полного статуса после N – 1 сообщений. Один из генералов все еще должен находиться в состоянии, в котором он не уверен, собирается ли другой атаковать.

Поскольку N – 1 сообщений будет недостаточно, N-е сообщение очень важно. Без него протокол не будет работать. И все же получение сообщения не гарантировано. Отправитель N-го сообщения не знает, будет ли оно получено. Поэтому отправитель N-го сообщения не имеет полного статуса и не получит его, так как в протоколе нет дальнейших сообщений. Эта ситуация показана на рис. 1.7.

---

[1] *Gray Jim.* (1978). Notes on Data Base Operating Systems. Chapter 3. Operating Systems, an Advanced Course. Springer-Verlag, London, UK.

[2] *Akkoyunlu E. A., Ekanadham K., Huber R. V. (*1975). Some constraints and tradeoffs in the design of network communications. Published in SOSP 1975. DOI:10.1145/800213.806523.


**Рис. 1.7.** Отправитель последнего сообщения не имеет полного статуса

Это противоречит вашему утверждению, что протокол предоставляет полный статус при конечном числе сообщений. Поэтому мы можем заключить, что такого протокола не существует.

## Смягчение ограничений

Проблема двух генералов (TGP) является аналогом многих проблем, которые мы пытаемся решить в распределенных системах. Используя только ненадежные сети для передачи сообщений между узлами, мы должны построить системы, которые тем не менее достигают соглашения с высокой степенью уверенности. Невозможность TGP, казалось бы, говорит нам о том, что это глупая затея. К счастью, проблемы, которые мы решаем в распределенных системах, немного проще, чем этот вымышленный аналог.

Рассмотрим банкомат. Клиент банка использует терминал, чтобы снять наличные со своего счета. Эта обычная повседневная операция кажется реальностью, созданной TGP. На западе у вас есть терминал банкомата, способный выдавать наличные. На востоке – центральный компьютер банка, который регистрирует движение денег на счетах клиентов. Между ними – враждебная территория цифровых коммуникаций, угрожающая прервать доставку сообщений.

Мы хотим, чтобы транзакция либо удалась, либо не удалась. Если она успешна, то деньги выдаются, и счет клиента дебетуется. В случае неудачи наличные не выдаются, и счет не дебетуется. Мы хотим избежать результата, в котором успех с одной стороны и неудача – с другой. Клиенты были бы очень расстроены, если бы их счета дебетовались, но наличные не поступали, а банки теряли бы деньги, если бы их банкоматы выдавали наличные без соответствующего списания.

## Переопределение проблемы

Невозможность решения TGP говорит нам, что это невозможно сделать. И все же миллионы транзакций банкоматов обрабатываются каждый день[1]. Очевидно, здесь что-то не так. В примере с банкоматом мы не смогли распознать, что ограничения на систему более мягкие, чем кажется на первый взгляд. Давайте рассмотрим подробнее причину невозможности решения TGP. Отсюда мы сможем увидеть, как ослабить ограничения и создать жизнеспособный протокол.

В первоначальном виде проблема имеет два строгих ограничения:

1) генерал не будет атаковать, если у него нет уверенности в том, что другой генерал тоже будет атаковать;
2) атака произойдет утром.

Согласно первому ограничению, поведение каждого генерала основано на том, что он знает о поведении другого генерала. До тех пор, пока один из генералов находится в состоянии неопределенности, оба остаются неопределенными. Не существует сообщения, которое могло бы одновременно изменить их мнение.

Согласно второму ограничению, существует крайний срок. Когда этот срок наступит, они должны достичь согласия. Любые сообщения, уже находящиеся в пути к этому моменту, не должны повлиять на окончательный результат. Не будет никаких дополнительных сообщений для устранения любой затянувшейся неопределенности.

Если мы ослабим эту пару ограничений, то сможем сформулировать задачу, которая имеет приемлемое решение. Действительно, можно найти протокол, который обменивается полным статусом, если разрешить одной стороне действовать в условиях неопределенности и убрать крайний срок. Это нарушает изложение проблемы двух генералов, но подходит для примера банкоматов. Действительно, мы обнаружим, что эта смягченная версия подходит ко многим бизнес-проблемам, которые решаются с помощью распределенных систем.

## Решать и действовать

Сначала ослабим ограничение, согласно которому генерал будет атаковать только в том случае, если он уверен, что его коллега тоже нападет. Западный генерал решает, что время пришло, и готовится к атаке независимо от того, что происходит на востоке. То, что является глупым поведением для генерала, может быть оправданным компромиссом для банкомата. Когда клиент снимает деньги со своего счета через банкомат, одна из сторон должна действовать без полной уверенности в том, что другая сторона последует ее примеру. Либо банкомат должен выдать наличные, либо компьютер банка должен зафиксировать списание. Рассмотрим последствия и шаги по исправлению каждого решения, если оно окажется односторонним.

Предположим, что банк регистрирует дебет, но банкомат не выдает наличные. В этом случае клиент покидает терминал без наличных, но банк считает,

[1] В исследовании Федеральной резервной системы США по платежам за 2013 год сообщается о 5,8 млрд снятий денег в банкоматах в 2012 году.

что деньги он получил. Следствием этого является то, что клиент недоволен, когда обнаруживает проблему, и его доверие к банку подорвано. Корректирующим действием является отмена дебета после обнаружения проблемы.

Теперь предположим, что банкомат выдает наличные, но банк не может зафиксировать дебет. В этом сценарии клиент ушел счастливым, а банкомат повторяет попытки коммуникации до тех пор, пока она не будет успешной. Тем временем, возможно, клиент сможет снять деньги в другом банкомате, поскольку банк не знает, что его баланс был исчерпан. Если это так, то корректирующим действием будет взимание с клиента комиссии за овердрафт.

Очевидно, что один из этих сценариев лучший как для банка, так и для клиента. Он защищает доверие, передает власть в руки клиента и дает банку дополнительный поток доходов. Итак, в этой ситуации разработчик распределенной системы определяет, что банкомат будет выдавать наличные, даже если неизвестно, зафиксирует ли банк дебет.

## Принять истину

Разработчик может уверенно принять это решение, только если он ослабит второе ограничение – что существует крайний срок. Предположим, что банкомат выдал наличные, но затем возникли технические трудности при сообщении этого факта банку. Может потребоваться некоторое время, пока техник починит терминал банкомата и восстановит канал связи. Когда терминал сообщает банку, что наличные были выданы, банк должен признать эту истину. Он не может отклонить транзакцию на основании прошедшего времени или текущего баланса счета клиента.

Повреждения банкомата могут быть настолько серьезными, что цифровая запись транзакции не может быть восстановлена. Возможно, произошел полный невосстановимый сбой жесткого диска. В этом случае можно прибегнуть к дополнительной экспертизе – подсчитать оставшиеся в банкомате наличные деньги и определить, была ли завершена последняя транзакция. Если банкомат, включая все наличные деньги, полностью уничтожен, то даже этот метод может оказаться недоступным. Но, конечно, в этом случае банк потеряет больше, чем одну транзакцию. Принятие истины означает принятие некоторого риска.

## Действенный протокол

Учитывая эти смягченные ограничения, мы можем теперь разработать протокол, который в конечном итоге достигнет полного статуса. Одна сторона (в данном случае банкомат) достигает точки, где она может уверенно принять решение. Она действует (выдает наличные), а затем продолжает протокол до тех пор, пока не узнает, что другая сторона знает о принятом решении. Он продолжает это делать независимо от того, сколько прошло времени или какие противоречивые обстоятельства вмешались.

Чтобы принять решение, банкомат связывается с центральным банком. Он проверяет, достаточно ли у владельца счета средств для выдачи запрашиваемых наличных. Он также проверяет свое местное хранилище банкнот, чтобы убедиться, что он сможет выполнить свою часть транзакции. В ходе этого

процесса банк может наложить временный арест на средства клиента. Но этот арест только снижает вероятность возникновения овердрафта, но не может его предотвратить. Банкомат со своей стороны накладывает временный арест на хранилище купюр – только один клиент может пользоваться банкоматом в данный момент. Если обе эти проверки пройдут, то банкомат выдаст наличные. Он принимает окончательное решение.

После принятия решения банкомат переходит ко второй фазе. На этом этапе решение уже принято, наличные выданы. Цель данной фазы заключается в том, чтобы сообщить об этом факте центральному банку. Вторая фаза не ограничена по времени, и истина не может быть опровергнута.

Этот вид протокола Джим Грей (Jim Gray) в 1978 году назвал *двухфазным действием* (Two Phase Commit – 2PC). На первой фазе, известной как фаза *голосования*, координатор получает от каждого участника подтверждение того, что он может зафиксировать запрошенную транзакцию. На второй фазе – фазе *фиксации* – координатор информирует каждого участника о своем решении. В предыдущем примере банкомат играет роль координатора и одного участника. Он является единственным лицом, принимающим решение после того, как он собрал достаточно информации для этого.

## Примеры неизменяемой архитектуры

Преимущества неизменяемости не остались не замеченными разработчиками распределенных систем. Некоторые из наиболее успешных распределенных систем, используемых сегодня, построены на этой концепции. Они получают от неизменяемости такие возможности, которых было бы трудно достичь в противном случае. Три примера – Git, блокчейн и Docker.

Git – это распределенная система контроля версий, популярная как среди команд разработчиков открытого исходного кода, так и среди корпоративных разработчиков. Она предлагает преимущество автономии для каждого отдельного разработчика. Разработчик может вносить изменения, переключаться между параллельными версиями продукта и разрешать конфликты – и все это в пределах изолированной реплики репозитория. Когда разработчики подключают свои реплики – напрямую друг с другом или с общим центральным хранилищем, – они только обмениваются информацией. Во время этого обмена не происходит блокировки или согласования, что делает взаимодействие быстрым.

Блокчейн – это термин для обозначения целого ряда родственных архитектур. Первым блокчейном был биткойн – распределенная валюта, полностью основанная на криптографических алгоритмах. Большинство блокчейнов сохраняют экономические аспекты валюты, но некоторые накладывают дополнительные функции на основную структуру данных. Главной особенностью блокчейна является неизменяемый *распределенный реестр*, обеспечивающий гарантию достоверности единой, прозрачной истории.

Docker – это технология выполнения программного обеспечения в контейнерах, как если бы вся операционная система и все зависимости были заключены в единую изолированную среду исполнения. Это эволюция за

пределы физических машин, которые действительно выполняли изолированную работу, и виртуальных машин, которые имитировали эту среду для целей переносимости и масштабирования. Docker достигает эффективности, которой не хватало виртуальным машинам, благодаря разумному использованию структурного разделения и неизменяемых образов дисков. Это привело к дальнейшему продвижению в области управления, например кластерам Kubernetes и mesh-вычислениям.

Все эти примеры используют преимущества неизменяемости для обеспечения своих основных возможностей. Интересно, что все они также являются открытыми системами. Скорее всего, это просто следствие того, что открытое программное обеспечение легкодоступно для анализа и поддержки в качестве архитектурных примеров. Нет никаких причин полагать, что неизменяемость не будет столь же ценной для закрытой системы, как и для открытой. Давайте проанализируем каждую из них немного подробнее, чтобы увидеть, как неизменяемость служит своим целям.

## Git

Git стремится предоставить каждому разработчику автономию, предоставляя всю необходимую информацию в реплике репозитория. Репозиторий состоит из отдельных изменений исходного кода, называемых коммитами. Коммиты неизменяемы и содержат ссылки на связанные с ними коммиты. Весь репозиторий – это постоянно растущая история коммитов, накопленная в течение жизни проекта.

Коммит – это набор изменений, внесенных в проект одним разработчиком в один момент времени. Идентичность коммита определяется исключительно его содержимым – именами используемых файлов, изменениями, сделанными в этих файлах, именем разработчика, а также причиной и временем внесения изменений. Это содержимое хешируется, и полученный хеш-код отныне используется для идентификации коммита. В результате получается неизменяемый граф коммитов, подобный показанному на рис. 1.8. Каждый разработчик, клонирующий репозиторий, будет вычислять один и тот же хеш для каждого коммита, что делает эти идентификаторы детерминированными и последовательными.

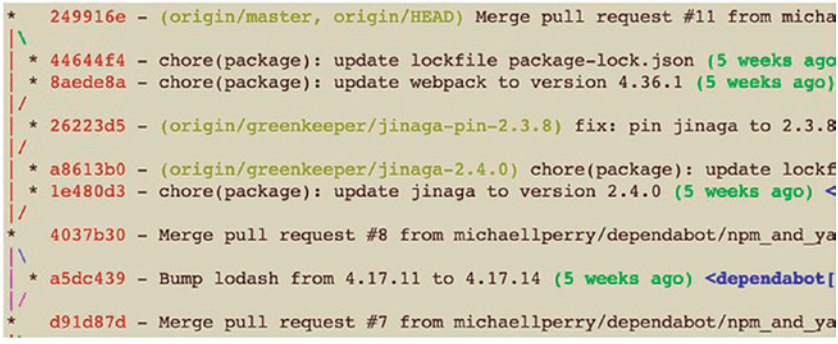


**Рис. 1.8.** История коммитов в репозитории Git

Текущее состояние исходного кода может быть построено из коммитов без повторного подключения к удаленному узлу, что обеспечивает автономность каждого узла. Разработчик работает, отключившись от сервера, чтобы самостоятельно создать новый набор коммитов. Пока коммиты работают, они не подключаются к удаленному узлу, чтобы заблокировать файлы или проверить наличие последних изменений. Они работают в полной изоляции, никаких обращений к серверу не требуется.

Когда возникает конфликт, как это часто бывает в исходном коде, разработчик находит в своем локальном хранилище всю информацию, необходимую для его разрешения. У него есть идентификационные данные сотрудников (возможно, даже самого себя), вовлеченных в конфликт. Он знает точный контекст изменения – как выглядел код в момент его изменения. И даже из комментариев к коммиту у него есть некоторые подсказки о намерениях каждого программиста.

На основе всей этой информации разработчики могут самостоятельно разрешить конфликт. Им не нужно привлекать сервер. На самом деле из-за природы ветвей Git они могут оставить конфликт в силе до тех пор, пока им это удобно. Нет необходимости, чтобы конфликт был разрешен, прежде чем можно будет продолжить работу. Но когда разрешение конфликта происходит, оно записывается в виде другого коммита. Этот коммит становится частью истории, чтобы все вовлеченные стороны могли видеть, что конфликт разрешен, и понимать эффект от его разрешения.

Такой режим работы возможен только потому, что каждый коммит неизменяем. Каждый разработчик, у которого есть такой же коммит, знает, что его копия так же хороша, как и любая другая. Ни один разработчик не может изменить содержимое или идентичность коммита. Все, что он может сделать, – это создавать новые коммиты.

## Блокчейн

Блокчейн хранит информацию в виде отдельных единиц (транзакций, контрактов, цифровых активов), объединенных в блоки. Блок – это просто коллекция этих единиц, окруженная оболочкой из метаданных. Каждый блокчейн определяет свою собственную структуру данных блока, но все они имеют некоторые общие поля:

- случайное число, называемое nonce;
- ссылка на предыдущий блок;
- хеш содержимого блока (включая nonce и ссылку).

В результате текущий блок – это просто самая последняя коллекция транзакционных единиц. Он указывает на предыдущий блок, который снова указывает на предыдущий блок, образуя цепочку. Эта цепочка, как показано на рис. 1.9, представляет собой всю историю транзакций с момента их создания.

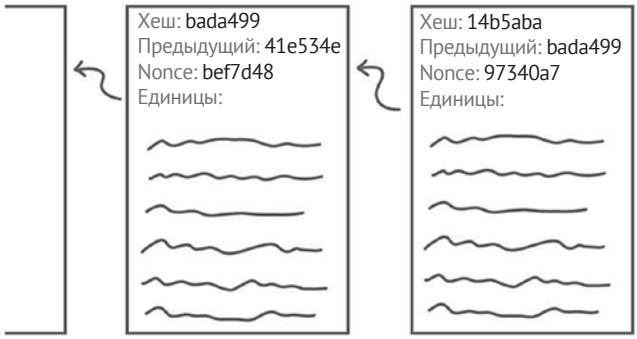


**Рис. 1.9.** Каждый блок в цепочке содержит хеш и ссылку на предыдущий блок

Неизменность блока является следствием хеша, который является его идентификатором. Если содержимое блока изменится, новый хеш будет другим. Криптографически сильные хеш-функции используются для того, чтобы было трудно изменить блок так, чтобы хеш остался неповрежденным. И когда я говорю здесь «трудно», я использую этот термин так, как его используют криптографы. Нам не разрешается говорить «невозможно».

Поскольку хеш блока (и, следовательно, его идентичность) включает хеш его предшественника, любые изменения в блоке пройдут через все последующие блоки и создадут новую альтернативную цепочку. Такая фальсификация будет легко обнаружена. Каждый узел видит одну и ту же копию каждого блока. Это одновременно и благоприятная характеристика, и наиболее ценная особенность блокчейна. С одной стороны, это позволяет немедленно обнаружить фальсификацию, а с другой – обеспечивает преимущество публично проверяемого реестра.

У блокчейнов есть недостаток, который, по мнению многих аналитиков (в том числе и моему), будет их гибелью. Чтобы гарантировать, что все узлы согласованы с одинаковой историей блоков, в большинстве блокчейнов используется *доказательство работы*. Это алгоритм, который медленно выполняется, но быстр для проверки. Например, блокчейн может потребовать, чтобы первые несколько битов хеша блока были равны нулю. Поскольку узлам приходится тратить такты процессора на вычисление доказательства работы, скорость добавления блоков в цепочку остается постоянной. Стоимость фальсификации цепочки – в смысле электроэнергии и вычислительного оборудования – больше, чем ценность, которая может быть получена. К сожалению, это означает, что стоимость *законного* использования блокчейна также очень высока по сравнению с его ценностью.

Хотя доказательство работы может оказаться «ахиллесовой пятой» блокчейна, выгода, которую он дает благодаря *неизменяемости*, очень весома. Только гарантируя, что каждый участник имеет одинаковую нестираемую копию реестра, эта система может обеспечить преимущества общей проверяемой цепочки.

## Docker

Docker выходит за рамки возможностей виртуальных машин, поскольку организует образы в *слои*. Слой – это неизменяемая часть файловой системы со ссылкой на слой ниже. *Образ* – это не что иное, как ссылка на самый верхний слой. Поэтому остальные слои также называют *промежуточными образами*.

Чтобы Docker мог выполнить работу, он создает *контейнер*. Контейнер – это запущенный экземпляр среды выполнения в комплекте с собственной смоделированной файловой системой. При запуске контейнера Docker выделяет ему специальный записываемый слой, который, в свою очередь, указывает на самый верхний слой образа. Изначально этот слой пуст.

Во время выполнения, когда контейнер читает с диска, Docker направляет эту операцию чтения в слой с возможностью записи. Поскольку этот слой изначально пуст, запрос на чтение попадет в самый верхний слой образа. Если запрашиваемые данные находятся в этом слое, то они будут получены. В противном случае запрос переместится в нижележащий слой.

Когда контейнер Docker записывает данные на диск, он изменяет только слой, доступный для записи. Этот слой является особенным, поскольку он не разделяется ни с одним из других контейнеров Docker и не сохраняется по окончании времени существования контейнера. Любая информация, записанная в этот слой, теряется при удалении контейнера. Даже если контейнер «перезаписывает» части операционной системы, нижний слой, содержащий исходные данные, не пострадает.

Идентификатор слоя представляет собой хеш, подобно идентификатору коммита Git или блока блокчейна. Разница, однако, в том, что это хеш не содержимого, а команды, которая его создала (включая любые исходные файлы в случае команды «Добавить» или «Копировать»). Результирующая структура показана на рис. 1.10. Чтобы создать образ, Docker начинает с базового образа – имени, используемого в реестре для идентификации хеша существующего слоя. Начиная с этого базового слоя, Docker затем сканирует команды одну за другой и вычисляет хеш полученного промежуточного образа. Если этот образ уже находится в репозитории, то он извлекается, а не реконструируется.

```
IMAGE          CREATED BY
3d53cdc8aa80   /bin/sh -c #(nop)  CMD ["pm2-runtime" "npm" …
8e48995a6b00   /bin/sh -c #(nop)  USER node
e022d570f832   /bin/sh -c npm install pm2 -g
277f48e981fd   /bin/sh -c npm install --production
77178a97af83   /bin/sh -c #(nop) COPY multi:56a9de3a81cb55e…
```

**Рис. 1.10.** Каждый образ Docker создается путем применения команды к предыдущему образу

Неизменяемые слои позволяют одному хосту Docker запускать несколько контейнеров из одного и того же или связанных образов без дублирования всей операционной системы для каждого из них. Виртуальные машины не могут совместно использовать образы, поскольку эти образы являются изменяемыми. Если одна машина изменяет образ, это изменение становится видимым для других виртуальных машин. Но Docker может реализовать совместное использование слоев, потому что эти слои не будут изменены.

Именно структурное разделение слоев позволяет Docker поддерживать управление взаимосвязанными запущенными контейнерами, сформированными из общего хранилища образов.

Каждая из этих систем использует возможности неизменяемости для обеспечения своих собственных преимуществ. Как отметил Пэт Хелланд в статье «Неизменяемость меняет все», эта идея является повторяющейся темой, появляющейся на нескольких уровнях стека технологий и во многих проблемных областях. Когда вы научитесь моделировать бизнес-проблемы на основе неизменяемости, вы начнете пользоваться преимуществами надежного аудита истории, как в блокчейне. А когда вы научитесь внедрять неизменяемые структуры данных в своих мобильных приложениях и микросервисах, вы получите преимущества автономии, которая есть в Git. Пусть путешествие начнется.

# Глава 2

# Формы неизменяемой архитектуры

Разработка системы, использующей только неизменяемые записи, имеет свои последствия. Некоторые из них – это те преимущества, которые мы уже рассмотрели: надежная связь, уменьшение блокирования, повышенная автономность и улучшенная проверяемость. Другие последствия менее желательны. Многие из них просто требуют изменения мышления, в то время как иные требуют совершенно новых решений. По мере внедрения неизменяемости в разработку приложений вам нужно будет осознать, как должна измениться их архитектура в ответ на это.

Компромиссы, требующие перехода к неизменяемости, привели к появлению различных архитектурных стилей. В этой главе мы рассмотрим три таких стиля: поиск событий (Event Sourcing – ES), обновление асинхронного представления модели и историческое моделирование. Все три стиля объединяет идея о том, что состояние развивается из исторических записей. В чем они расходятся, так это в упорядочивании этих записей. Первые два стиля предполагают, что записи можно просматривать *последовательно*, и они могут быть перечислены по порядку. Третий возникает из идеи, что исторические записи могут быть *частично* упорядочены. Он не допускает перечисления. Вместо этого он пользуется данной возможностью для достижения некоторых ценных результатов.

После данной главы остальная часть книги будет посвящена третьему стилю – историческому моделированию. Но будет важно вписать этот выбор в контекст. Каждое архитектурное решение – это компромисс между конкурирующими ценностями. Давайте рассмотрим все три архитектуры, чтобы лучше понять, какими будут эти компромиссы. Начнем мы с общих для всех них концепций.

## Выведение состояния из истории

Искусство неизменяемой архитектуры заключается в нахождении баланса. С одной стороны, неизменяемые структуры данных дают значительные преимущества в параллельных и распределенных вычислениях. С другой – мир, который моделируют наши системы, полон изменений. Каждая из архитектур, которые мы будем изучать, находит этот баланс по-своему.

Это помогает дать вещам четкие и осмысленные названия. Мы будем называть вещи, которые изменяют мир, *объектами*, а вещи, которые не изменяют, – *записями*. Этот выбор не произвольный. Он основан на понятиях, которые люди придумали для организации информации как с помощью компьютеров, так и без них.

## Исторические записи

Давайте вернемся в прошлое, в мир до появления компьютеров. Как велись дела в этом мире? Вместо того чтобы обновлять текущее состояние мира в большой базе данных, информация записывалась и передавалась в форме документов.

Предположим, клиент размещает заказ на десять виджетов. Это решение фиксируется в виде заказа на покупку. В заказе на покупку упоминаются две стороны: покупатель и продавец. В нем также содержатся ссылки на товар – виджеты – по каталожному номеру, присвоенному продавцом. В результате получится документ, подобный показанному на рис. 2.1.


**Рис. 2.1.** Заказ на покупку фиксирует намерение клиента приобрести товар у поставщика

### Опираясь на прошлое

Заказ на поставку – это историческая запись. Это неизменяемый документ, который фиксирует решение, каким оно было в определенный момент времени. В нем упоминаются две стороны: покупатель и продавец. Ни одна из сторон не может изменить заказ. Они могут только заменить этот документ другим.

Покупатель и продавец являются отдельными юридическими лицами. Эти субъекты были созданы с собственным набором неизменных записей – документов, которые были поданы в соответствующие регулирующие органы в качестве учредительных документов. Эти документы были созданы задолго до заказа на покупку.

Заказ на покупку связан с несколькими другими решениями, которые были приняты до него. Например, он ссылается на каталожный номер. Это результат решения о публикации виджета в каталоге доступных продуктов с указанием цены. Каталог после публикации не изменяется. Он изменяется только путем публикации последующих каталогов с постоянно обновляющимися списками товаров и цен.

### Развитие понимания

Будущие документы, в свою очередь, будут ссылаться на этот заказ. Как только продавец получит копию этого документа, он создаст счет-фактуру – новый документ, требующий оплаты. В ответ на это покупатель выписывает документ – чек, в котором просит банк перечислить средства продавцу. Продавец создает упаковочную накладную, в которой указываются товары, которые должны быть включены в груз. Перевозчик выписывает накладную покупателю, документируя доставку товара.

Исторические записи развивают наше коллективное понимание с течением времени. Каждая из них сама по себе неизменна. Но наш взгляд на мир меняется по мере того, как публикуется все больше документов.

## Изменяемые объекты

Изобретение компьютера значительно расширило наши возможности по обработке транзакций. Но оно также неуловимо изменило то, как мы думаем о том, что является истиной. До появления компьютеров каждому из нас приходилось носить с собой банковскую книжку, чтобы подсчитать баланс своего счета. Сейчас мы можем сразу же увидеть его на своем мобильном телефоне. Раньше считалось, что идея «баланса» была производной и отличалась в зависимости от знания того, какие чеки были обналичены. Теперь, по крайней мере интуитивно, «баланс» стал неотъемлемым свойством объекта «счет».

Как и функциональное, модульное и структурированное программирование до нее, *объектная ориентация* была одним из величайших достижений программного моделирования. Она возникла из наблюдения, что программные системы существуют для моделирования реального мира. Она стремилась сначала понять поведение моделируемых объектов, а затем предоставить образцы и шаблоны для реализации.

В своей основе объектно-ориентированное программирование предполагает, что по крайней мере *некоторые* объекты являются изменяемыми. Оно оставляет место для неизменяемых объектов, но в основном описывает поведение и эволюцию изменяемых объектов. Это предположение проявляется прежде всего в концепции идентичности.

### Идентичность

Мы часто говорим о трех фундаментальных элементах объектной ориентации, которые Джеймс Румбо (James Rumbaugh) определил в 1991 году. Это инкапсуляция, наследование и полиморфизм. Единственное, о чем мы не упоминаем, – это *идентичность*. Румбо определяет идентичность как отличительное свойство, не зависящее от идентифицирующего атрибута. В объектно-ориентированном моделировании, даже если два объекта имеют одинаковые свойства, они являются разными объектами. Действие на один из них не повлияет на другой.

> Два объекта различны, даже если все значения их атрибутов (таких как имя и размер) идентичны[1].

[1] *James Rumbaugh et al.* Object-Oriented Modeling and Design. 1991 Prentice-Hall, Inc. ISBN 0-13-629841-9.

Когда мы переносим эту идею на объектно-ориентированные языки, такие как Java или C++, понятие идентичности сопоставляется с местоположением в памяти. Объект становится блоком памяти, выделенным для хранения его текущего состояния. Идентичность объекта – это адрес в этой памяти. Вот почему совместное использование идентичности объекта в C++ достигается путем «передачи указателя».

Адреса памяти подчиняются правилам объектно-ориентированной идентификации. Они представляют уникальность без каких-либо идентифицирующих атрибутов. В любой момент времени состояние двух объектов может быть совершенно одинаковым. Куски памяти по этим двум адресам могут быть байт за байтом одинаковыми. Но изменение памяти в одном месте не окажет никакого влияния на память в другом месте. Потребитель одного объекта не воспримет никаких изменений в его поведении на основе модификации другого.

По мере того как мы все дальше уходим от одного потока в одиночном процессе на изолированной машине, это решение начинает показывать свои недостатки. При переходе от одного потока ко многим мы должны ввести блокировку для защиты целостности структуры данных от одновременного изменения. Переходя от одного процесса ко многим, мы должны отобразить используемый общий объект в независимые пространства памяти. А если выйти за пределы одной машины, то понятие указателя теряет всякий смысл. Необходимо ввести другие формы идентификации, чтобы компенсировать это.

### Изменение состояния

Определение идентичности, данное Румбо, решает проблему для объектов, которые могут менять состояние с течением времени. Но если мы вводим понятие неизменяемости, оно становится менее полезным. Причина, по которой объект должен обладать внутренней идентичностью, заключается в том, что он может обеспечить последовательное, осмысленное поведение, по мере того как он изменяется во времени. Если я откушу от одного яблока, другое останется целым. Это был бы очень странный мир, если бы в *вашем* яблоке появился след от укуса или если бы я вернулся позже и обнаружил, что мое яблоко полностью восстановлено.

Идентичность Румбо – это признание того, что объекты в реальном мире меняют состояние в ответ на воздействие. Они помнят это состояние. Их наблюдаемое поведение основано на их текущем состоянии. Чтобы эти изменения в поведении имели смысл в любой модели реального мира, объекты в модели должны обладать внутренней идентичностью.

## Проекции

Сейчас наша цель – использовать неизменяемые записи для моделирования изменяемых объектов. Записи явно не являются самими объектами. Этого недостаточно, так как записи не будут позволять изменять объекты, которые, как ожидается, могут изменяться. Вместо этого записи должны каким-то образом представлять *изменения* в объектах. Неизменяемые записи *являются* изменениями объектов.

Чтобы достичь этой цели, мы будем рассматривать неизменяемые записи как *наблюдаемое* состояние. Они представляют вещи, которые мы действительно видели и записали. Объекты, с другой стороны, являются *производным* состоянием. Они представляют нашу интерпретацию этих наблюдений и могут меняться по мере новых наблюдений.

## Два вида состояния

Представьте себе электронную таблицу. В каждую ее ячейку можно ввести либо значение, либо формулу. Значение представляет собой некоторое базовое измерение, наблюдение за системой, которую вы моделируете. Формула, с другой стороны, выводит новое значение из этих наблюдений. Формулы представляют собой производные состояния.

В математике аналог этой идеи – функция. Мы часто пишем y как функцию от x, создавая тем самым график. Вы можете провести вертикальную линию в любом месте этого графика, и попадете только в одну точку. То же самое нельзя сказать о любой горизонтальной линии. Мы говорим, что x представляет собой *независимую* переменную, а y – *зависимую* переменную. Вы можете выбрать x, а y вычисляется с помощью функции. Независимые переменные наблюдаются, а зависимые переменные выводятся.

В программировании есть другие названия для этого явления. Производное состояние иногда называют *проекцией* наблюдаемого состояния. Чистая функция принимает значение на вход и производит выход без побочных эффектов. Выход детерминирован и зависит только от входных данных. Если входом является наблюдаемое состояние, то выход получается из этого входа. Он является проекцией этого наблюдаемого состояния.

Производное состояние также появляется в программном обеспечении в виде *модели представления*. В то время как модель – это объект, который в целом поддерживает проблемную область, модель представления отображает более конкретно представление. Она проецирует модель для отображения на представлении. В системах связывания данных изменения, внесенные в модель представления, появляются непосредственно в представлении.

## Проецирование объектов

Не важно, как вы это называете – формулами, зависимыми переменными, проекциями или моделями представления, – производное состояние является детерминированным преобразованием наблюдаемого состояния. Оно не добавляет никакой информации в систему, а лишь представляет уже имеющуюся информацию в другом виде. В математике мы говорим, что оно не добавляет новых степеней свободы в систему. В программном обеспечении можно сказать, что модель представления поддерживается моделью. Важным моментом является то, что пользователь может непосредственно изменять наблюдаемое состояние. Он может видеть результаты, косвенно спроецированные на производное состояние.

В неизменяемой архитектуре исторические записи являются наблюдаемым состоянием. Пользователь получает возможность создавать новые записи непосредственно своими действиями. Эти записи фиксируют принятые пользователем решения.

С другой стороны, объекты – это всего лишь проекции. Они эфемерны. Пользователь не имеет права устанавливать состояние объекта. Он может только видеть, как эти объекты изменяются в результате новых исторических записей. Каждая из данных архитектур имеет свой собственный способ вычисления этой проекции.

## Поиск событий

Исторические записи – это наблюдаемое состояние неизменяемой архитектуры. Они представляют прошлые решения. Вы можете назвать эти прошлые решения «событиями» и потребовать, чтобы они были единственным источником истины. Таково происхождение термина *поиск событий* (event sourcing, ES).

Хотя термин «поиск событий» может быть применен к любой архитектуре, которая восстанавливает состояние из истории неизменных записей, эта практика несколько более специфична.

> Мартин Фаулер (Martin Fowler) в 2009 году описал ее так: «Фундаментальная идея поиска событий заключается в том, чтобы гарантировать, что каждое изменение состояния приложения фиксируется в объекте события и что эти объекты событий хранятся в той же *последовательности, в которой они были применены*, в течение того же времени жизни, что и состояние самого приложения»[1].

Акцент на последовательности их применения – мой. Идея последовательности не обязательно следует из требований неизменности исторических записей. Но это разумное предположение, и его разделяют все реализации ES, которые я видел. Поэтому я считаю *последовательность* определяющей характеристикой поиска событий.

### Генерация событий

В приложении, основанном на событиях, пользователь взаимодействует (через пользовательский интерфейс и, возможно, API) с моделью домена. Модель домена не отвечает на запрос немедленно. Вместо этого она проверяет запрос и генерирует событие. Событие – это неизменяемая запись о намерениях пользователя. Оно называется и интерпретируется как утверждение в прошедшем времени, как «это событие произошло», например OrderSubmitted (Заказ отправлен), PlayerRegistered (Игрок зарегистрировался) или ResidentMoved (Житель переехал). Соглашение об именовании отражает истину, что однажды сгенерированное событие нельзя проигнорировать. Его эффект может просто отличаться от того, что задумал пользователь.

Взаимодействуя с моделью домена, пользователь воспринимает приложение так, как будто оно следует традиционной объектно-ориентированной парадигме. У него создается впечатление, что объекты имеют свойства, которые изменяются с течением времени, и что его действия непосредственно вызывают эти изменения. Приложение скрывает тот факт, что объектная мо-

[1] *Martin Fowler*. https://martinfowler.com/eaaDev/EventSourcing.html.

дель является одновременно генератором и проекцией последовательности событий.

Преимущества, которые ES обеспечивает по сравнению с традиционной объектной моделью, начинаются с тех же самых, которые мы уже определили для всех неизменяемых архитектур, – повышенная масштабируемость и возможность проверки. Кроме того, они могут похвастаться способностью полностью перестраивать объекты из потока событий. Когда исправляется дефект или добавляется особенность, приложение может отбросить все кешированные версии доменной модели и восстановить их с помощью нового кода. Это также позволяет приложению, основанному на событиях, вернуться назад во времени и воспроизвести только часть последовательности, увидев объект таким, каким он был в прошлом. Пользователю приложения это дает мощную возможность проводить временной анализ.

Практики часто используют поиск событий вместе с разделением ответственности запросов и команд (Command Query Responsibility Segregation – CQRS) и проектированием, ориентированным на домен (Domain-Driven Design – DDD). Такое сопряжение не является обязательным требованием для ES, как и не все реализации согласованы с тем, как это достигается. Некоторые предпочитают использовать только CQRS в паре с ES или только DDD в паре с ES. Это архитектурное решение влияет на то, как приложение проецирует неизменяемые записи в изменяемые объекты.

## CQRS

Разделение ответственности команд и запросов расширяет объектно-ориентированный принцип разделения команд и запросов (Command Query Separation – CQS). Бертран Мейер (Bertrand Meyer) определяет команды и запросы как разновидности методов. Он различает их следующим образом:

> «Команда служит для модификации объектов, запрос – для возврата информации об объектах»[1].

Там, где CQS проводит границу между методами, CQRS расширяет эту границу до разделения объектов. Повинуясь принципу единой ответственности[2], некоторые объекты отвечают за выдачу команд, а другие – за выдачу запросов.

В CQRS *команды* отвечают за изменение состояния системы. Они отличаются от *запросов*, которые запрашивают информацию о текущем состоянии. Команды и запросы следуют по различным путям и часто взаимодействуют с различными архитектурными компонентами. Команды часто асинхронны, в то время как запросы обычно синхронны. Во многих реализациях они работают с разными хранилищами данных.

В паре с ES команды еще больше отличаются от *событий*. В то время как команда выражается императивным высказыванием, то событие – это выска-

---

[1] *Bertrand Meyer*. Object Oriented Software Construction, Second Edition. Prentice Hall. 1997. ISBN: 0-13-429155-4.

[2] *Robert C. Martin*. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. 2003. ISBN 978-0135974445.

зывание в прошедшем времени. Команда SubmitOrder (Отправить заказ) приводит к событию OrderSubmitted (Заказ отправлен). Команда инструктирует систему записи выполнить намерение пользователя. Событие, с другой стороны, *производится* системой записи и должно быть выполнено. Система записи отвечает за первую проверку и авторизацию команды. Она может выбрать отказ или игнорировать сообщение.

Когда команда посылается асинхронно, это еще больше отвлекает пользователя от последствий своих действий. Приложения, использующие архитектурный стиль CQRS/ES с асинхронными командами, часто демонстрируют пользователю *в конечном итоге последовательное* взаимодействие, давая ему понять, что его запрос будет обработан позже.

## DDD

Когда объектно-ориентированное программирование было впервые представлено, оно обещало, что объекты в программном обеспечении смогут моделировать объекты в мире. По мере его внедрения в разработку корпоративного программного обеспечения объекты стали меньше моделировать мир и больше – компьютер. Эрик Эванс (Eric Evans) перенаправил объектно-ориентированный анализ и проектирование на область проблем в своей книге 2004 года «Проектирование с учетом доменов» (Domain-Driven Design)[1].

Хотя в книге даются советы по многим фазам процесса разработки программного обеспечения, мы сосредоточимся только на технических аспектах, в частности на онтологии и взаимоотношениях различных видов объектов.

DDD распознает два вида объектов: сущности и типы значений. *Сущность* – это объект, который обладает идентичностью. Как мы уже видели, объектно-ориентированная идентичность позволяет объекту изменяться с течением времени. В отличие от этого, *тип значения* не имеет идентичности и поэтому является неизменяемым.

Клиент, например, является сущностью, поскольку обладает идентичностью и может меняться с течением времени. У клиента может быть изменяемое свойство ShippingAddress (Адрес доставки), тип данных которого будет Address (Адрес). Адрес, в свою очередь, может иметь несколько различных свойств, таких как улица (Street), город (City), страна (Country) и др. Но Адрес доставки управляется как единое целое, а не как отдельные свойства клиента. Тип данных Address является типом значения и не имеет собственной идентичности.

Сущности в DDD организованы в иерархии, называемые *агрегатами*. Агрегат построен на отношении «родитель–потомок». На вершине иерархии находится *корень агрегата*. Например, сборка (Assembly) в одном домене может быть агрегатом, содержащим множество частей (Part). Несколько сборок собраны под одним корнем агрегата, называемым продуктом (Product). Агрегат показан на рис. 2.2.

[1] *Eric Evans*. Domain Driven Design: Tackling Complexity in the Heart of Software. AddisonWesley Professional. 2004. ISBN: 0-321-12521-5.

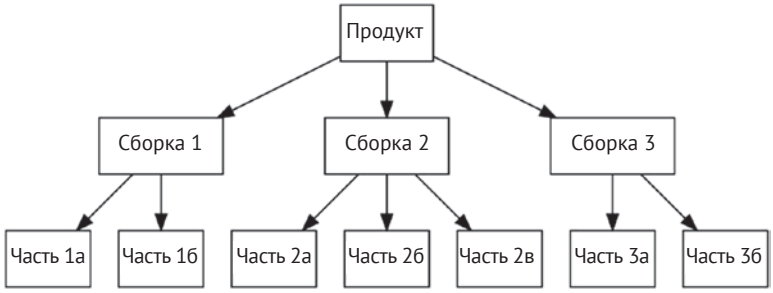


**Рис. 2.2.** Корень агрегата связан со своими дочерними сущностями

Как правило, к сущностям не обращаются вне корня их агрегата. В рамках домена, который мы описали ранее, не имеет смысла говорить о части без предварительного определения ее продукта. Продукт является корнем агрегата и, следовательно, точкой входа для всех внешних ссылок.

Поиск событий воспроизводит прошлые события, для того чтобы перестроить объекты в модели. Но это не означает, что система перестраивает всю объектную модель при каждом запросе. Это не является эффективным способом обработки запроса к одному объекту. Большинство событий в истории модели не будут иметь никакого влияния на этот объект. Поэтому вместо этого приложения, основанные на событиях, разбивают эту историю на независимые потоки. Каждый поток влияет только на подмножество доменной модели. Когда ES сочетается с DDD, это подмножество является корнем агрегата.

Запрос в приложении DDD/ES определяет корень агрегата. Приложение загружает поток событий для данной идентичности. Поток содержит все события, которые повлияли на корень или любую другую сущность в агрегате. Отдельные сущности не восстанавливаются из своей собственной истории, а возникают из истории их корневой сущности.

## Взгляд с точки зрения функций

До сих пор мы концентрировались на объектно-ориентированном анализе для описания действия событий. Но существует не менее обоснованная интерпретация с использованием идей функционального программирования. Рассмотрим функцию, которая вычисляет состояние системы после применения события:

$$f(state_n, event) \rightarrow state_{n+1}$$

Под состоянием системы мы можем понимать отдельный объект, корень агрегата или даже всю объектную модель. Практически говоря, вы захотите выбрать меньшую границу. Но аналитически результат одинаков.

Преимущество моделирования состояния системы с помощью этой функции заключается в том, что состояние, как и события, становится неизменяемым. Функция является чистой функцией – она не вызывает побочных эффектов. Более конкретно: она не изменяет состояние своих входов. Функция *не изменяет* входящее состояние, а возвращает новое состояние, *полученное* из входящего состояния.

В функциональном программировании нередко определяются функции высшего порядка. Это функции, которые принимают функции в качестве параметров. Одной из таких функций является сворачивание слева (`left-fold`, иногда ее сокращенно называют `foldl`). `foldl` берет бинарную операцию – функцию, принимающую два параметра, и применяет эту операцию к каждому элементу списка.

Например, имея бинарную операцию `+`, начальную точку `0` и список чисел, `foldl` вычислит сумму.

```
foldl(+,0,[3,17,2])=22
```

Вы можете думать о поиске событий в этих функциональных терминах. Бинарная операция – это функция, описанная ранее, которая принимает текущее состояние и возвращает состояние после применения события. Начальная точка – это начальное состояние системы. А список – это последовательность событий. Грег Янг (Greg Young) использует функциональные конструкции для описания поиска событий:

> «Когда мы говорим о поиске событий, текущее состояние является операцией сворачивания слева предыдущего поведения»[1].

## Коммутативные и идемпотентные события

Мы обнаружим, что свойства коммутативности и идемпотентности полезны в распределенных системах. Свойство коммутативности позволяет нам изменить порядок применения операции и получить тот же результат. Свойство идемпотентности дает нам возможность повторить операцию без дальнейшего эффекта. Поскольку поиск событий основан на последовательности операций, он чувствителен как к порядку, так и к дублированию. Разработчик приложения должен убедиться в том, что порядок сохраняется, и предотвратить дублирование.

Оператор `+` является коммутативным – `a+b = b+a`. Но в общем случае бинарные операции не должны быть коммутативными. Если вы не будете очень осторожны в выборе оператора, `foldl` будет чувствителен к порядку элементов в списке. Таким образом, поиск событий по умолчанию является некоммутативным. Если система должна последовательно реагировать на неупорядоченные события, то разработчик обязан доказать, что функция «событие-приложение» коммутативна, когда это необходимо. Точнее говоря, если функция «событие-приложение» удовлетворяет следующему уравнению, то события коммутативны:

```
f(f(x,e1),e2)=f(f(x,e2),e1)
```

Как не все бинарные операторы коммутативны, так и не все *идемпотентны*. Оператор `+` является одним из таких примеров `-a+a ≠ a` (за исключением специального случая, когда `a = 0`). И поэтому операция сворачивания слева над последовательностью, содержащей одинаковые числа, приведет к раздуванию результата. Если ваша распределенная система допускает дублирование событий в потоке, то вам предстоит доказать следующее уравнение:

[1] *Greg Young*. http://codebetter.com/gregyoung/2013/02/13/projections-1-the-theory/.

```
f(f(x,e),e)=f(x,e)
```

Вы можете обрабатывать порядок и дублирование перед потоком событий или после него. Либо предотвратить попадание неупорядоченных и дублирующих событий в поток, либо тщательно выбрать свою функцию применения событий.

## Асинхронное обновление представления модели

Язык программирования Elm воспринимает функциональное представление истории событий совершенно буквально. Этот язык компилируется в JavaScript и запускается в браузере. Он генерирует HTML из исходной модели. Чистая функция создает последовательные версии модели по мере обработки сообщений. Шаблон, на котором основан Elm, называется *Model View Update*.

На основе Elm фирма Facebook создала набор инструментов, которые распространяют этот шаблон на сервер. Первым из этих инструментов был React, внешняя библиотека для JavaScript, которая проецирует модель в HTML. Следующим был Flux – однонаправленный поток данных, который является шаблоном проектирования приложений. Это определило то, как Facebook разрабатывает веб- и мобильные приложения. Redux – это реализация Flux, разработанная вне Facebook Дэном Абрамовым (Dan Abramov). Дэн впоследствии был принят в Facebook для продолжения работы над Redux, React и архитектурой в целом. Наконец, есть внутренняя (back-end) архитектура, только часть которой в настоящее время имеет открытый исходный код, на основе которой Facebook разрабатывает свои API.

### Цикл обновления

Когда React и Redux используются вместе, они образуют цикл. Этот цикл является основным механизмом шаблона Model View Update. React преобразует модель в представление, а Redux выполняет действия по обновлению модели.

*Модель* (`model`) – это просто структура данных. Как разработчик приложения вы определяете структуру данных, которая вам нужна. Поскольку React ориентирован на одностраничные веб-приложения, модель обычно представляет собой коллекцию состояний, которые просматривает и которыми манипулирует один пользователь в пределах страницы.

*Представление* (`view`) – это то, что видит пользователь, в React – это HTML. Функция под названием `render` преобразует модель в представление. Эта функция запускается сначала после загрузки модели, а затем каждый раз, когда создается новая версия модели. Полученные результаты сравниваются, чтобы определить, что на самом деле изменилось, и обновить объектную модель документа (DOM). Разработчику не нужно заботиться об отслеживании изменений.

```
render(model) → view
```

Наконец, `update` – это функция, которая вычисляет следующую версию модели. Это чистая функция, поэтому она не изменяет модель. Вместо этого она представляет модель в том виде, как она будет выглядеть после совершения *действия*.

```
update(modeln,action) → modeln+1
```

Приложение Model View Update начинается с загрузки модели. Затем оно входит в цикл, в котором взаимодействие с пользователем генерирует действие. Функция `update` обрабатывает действие, создавая новую версию модели. Функция `render` превращает эту новую модель в новое представление. Фреймворк сравнивает две версии представления, чтобы определить, что нужно изменить в браузере. Пользователю представляется обновленный пользовательский интерфейс, и цикл продолжается, как показано на рис. 2.3.

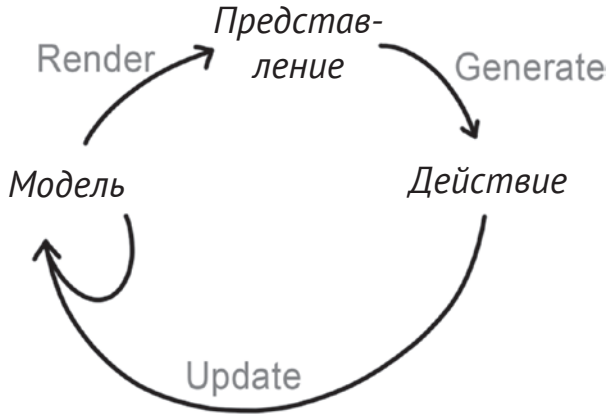


**Рис. 2.3.** Цикл «модель–представление–обновление», используемый в React и Redux

Будучи чистой функцией, `update` не изменяет модель. Она создает новую версию модели, которая затем отображается для создания новой версии представления. Поэтому правильнее изобразить этот цикл не как цикл, а как спираль, как показано на рис. 2.4.

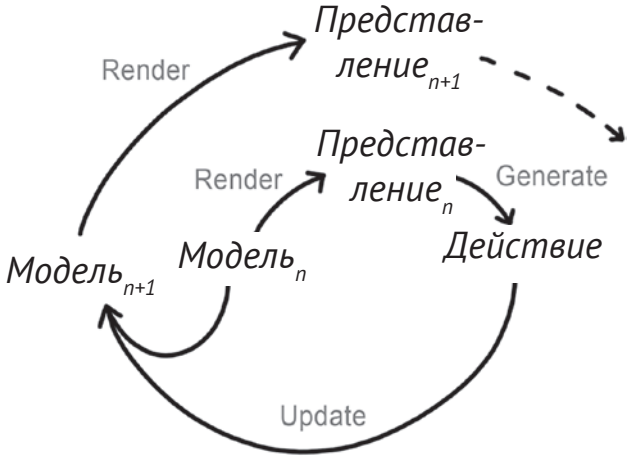


**Рис. 2.4.** Каждая итерация создает новую версию модели

Это изображение делает более наглядным отсутствие циклических зависимостей. Каждая итерация производит новый набор объектов. Старые объекты все еще доступны для поддержки оптимизации, например для минимизации манипуляций с DOM.

## Однонаправленный поток данных

Flux и Redux были разработаны как реакция на проблемы, обнаруженные в шаблоне *Model View Controller* (MVC). В MVC контроллер реагирует на изменения в модели путем обновления представлений. Он также реагирует на пользовательский ввод в представлении, обновляя модель. Контроллер координирует поток данных в двух направлениях: как внутрь от пользователя, так и наружу из приложения.

Сначала двунаправленный поток данных очень прост. Имея всего несколько контроллеров, несложно проследить поток выполнения от представления к модели и обратно.

Но по мере добавления контроллеров количество путей значительно увеличивается, как показано на рис. 2.5. При этом становится сложно определить, приведет ли новая функция к появлению нового крайнего случая, который вызовет каскадные обновления или циклические зависимости.

С другой стороны, при однонаправленном потоке данных отсутствует вероятность каскадного обновления. Действие производит обновление состояния, которое производит обновление представления. Представление не может ответить на одно действие, произведя другое действие. Не существует возможности циклов или бесконечных обновлений.

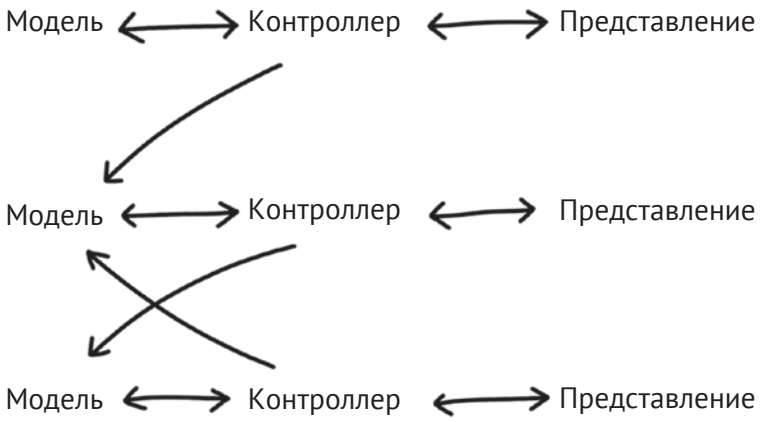


**Рис. 2.5.** Несколько контроллеров, координирующих друг с другом обновление своих моделей

Однонаправленный поток данных также поддерживает лучшее модульное тестирование. Начните с заданного состояния. Когда действие обработано, обработчик создает новое ожидаемое состояние. Инициализация состояния, применение действия и проверка результирующего состояния – все это легко автоматизировать. Операции, которые трудно автоматизировать, – проверка правильности отображения представления и что взаимодействие с пользователем правильно интерпретировано – отходят на второй план. Рендеринг

представления – это просто функция модели, а пользовательский ввод производит только действие. Однонаправленный поток данных минимизирует объем ручного тестирования.

## Неизменяемая архитектура приложений

Обновление модели и представления, практикуемое в React и Redux, – это только половина картины. Другая половина происходит на сервере. Хотя Redux имеет возможность работать с хранилищем в памяти, когда мобильное приложение взаимодействует с сервером, текущее состояние модели не всегда доступно. Ли Байрон (Lee Byron) представил решение Facebook на конференции Render 2016[1]. Он назвал его «Неизменяемая архитектура приложений». Но поскольку это название похоже на общий термин «неизменяемая архитектура», как я его использую, я буду называть представленный им шаблон *асинхронным обновлением представления модели*.

Шаблон начинается, как и раньше, с функции `render`, проецирующей модель в представление. Так же, как и раньше, взаимодействие пользователя с представлением порождает действия. Однако на этом этапе шаблон расходится. На стороне клиента действия передаются в *очередь*. На стороне сервера действия применяются к *истинному состоянию*. Функция `update` объединяет последнее известное истинное состояние со всеми действиями, все еще находящимися в очереди. Результирующий цикл показан на рис. 2.6.

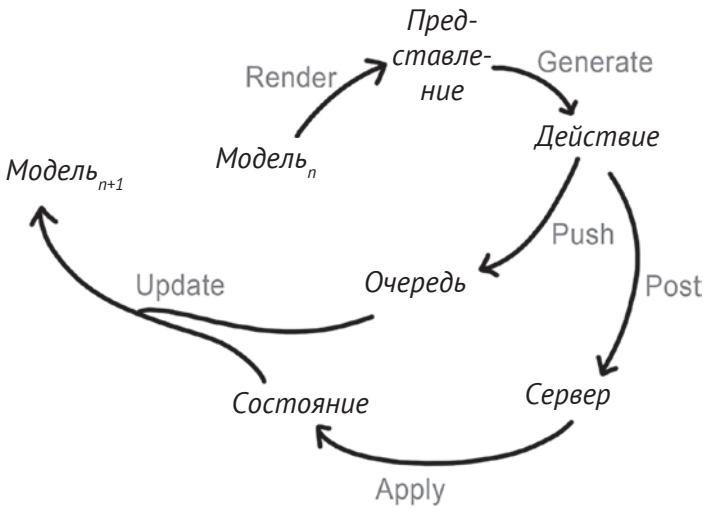


**Рис. 2.6.** Асинхронное обновление представления модели

Следующая итерация модели не основывается на предыдущей итерации. Вместо этого клиент возвращается к последнему состоянию, полученному от сервера. Если новое состояние получено от сервера асинхронно, то оно становится новым истинным состоянием. Любые действия, которые представлены

[1] *Lee Byron.* Immutable User Interfaces. Render 2016. https://vimeo.com/166790294.

в этом состоянии, удаляются из очереди. Очередь содержит действия, которые могут стать истинными.

Преимущество этой архитектуры для Facebook заключалось в том, что стало легче добавлять новые функции. Чтобы добавить функцию, разработчику нужно было только добавить новые действия и распространить их на представление. Если эти действия влияли на другие представления, разработчик просто добавлял нужный эффект в функцию обновления. Мобильное приложение при этом может работать быстро, даже при медленной сети. Эффект от действия пользователя виден сразу, без необходимости ожидания реакции сервера.

Архитектура асинхронного обновления представления модели оптимистично интерпретирует серию действий. Действия пользователя проверяются на устройстве с расчетом на то, что большинство из них будут успешными на сервере. Предполагается, что никакие другие действия не будут вмешиваться и что результат выполнения действий на сервере будет таким же, как и на клиенте. Когда это оптимистическое предположение оказывается ложным, архитектура просто отбрасывает локально вычисленное состояние и берет версию сервера.

## Историческое моделирование

Неизменяемые архитектуры, которые мы только что рассмотрели, проводят различие между неизменяемыми историческими записями и изменяемой объектной моделью. Они также предполагают, что исторические события происходят в полностью упорядоченной последовательности. Но ни одно из этих предположений не следует обязательно из идеи использования неизменяемых записей в качестве источника истины. Если смоделировать систему как набор связанных исторических фактов, можно полностью отказаться от модели изменяемых объектов, и факты при этом не обязательно должны происходить в последовательности.

Начнем с небольшого изменения терминологии. Вместо того чтобы ссылаться на исторические записи как события или действия, давайте называть их фактами. Причина изменения названия заключается в том, что факты подчиняются набору правил, которые не обязательно применяются к событиям в поиске событий или действиям в асинхронном обновлении представления модели. В частности, факты частично упорядочены.

### Частичный порядок

Термин «частичный порядок» пришел из математики и отличается от термина «полный порядок». Начнем с набора объектов, будь то числа, слова, научные работы, структуры данных и т. д. Определим операцию сравнения, которая скажет вам, находится ли один объект перед другим. Обычно мы будем использовать символ «меньше чем» (<) для представления этой операции. Если для любой пары объектов в наборе мы можем использовать <, чтобы поставить один объект перед другим, то набор полностью упорядочен. Если мы можем сделать это только для некоторых пар, то набор упорядочен частично.

В качестве примера рассмотрим множество счетных чисел и знакомое определение оператора <. 1<3 и 3<17. На самом деле для любой пары разных счетных чисел можно использовать этот оператор, чтобы поставить одно перед другим. Если a и b различны, то либо a<b, либо b<a. Множество полностью упорядочено по оператору <.

Но теперь давайте изменим определение < и посмотрим, что произойдет. Вместо привычного «меньше чем» давайте скажем, что a<b, если a является простым множителем b. То есть если a и b различны и деление b на a не оставляет остатка, то a<b. Теперь мы можем сравнить пары чисел и посмотреть, что получится. 3<15, потому что 3 является простым множителем 15. Но мы не можем сказать, что 3<8 или что 8<3. Ни одно из них не является множителем другого. Таким образом, счетные числа частично упорядочены под действием соответствующего фактора.

Говорим ли мы о полном или частичном порядке, оператор сравнения, который мы выбираем, должен обладать парой полезных свойств. Во-первых, он должен быть транзитивным.

a<b и b<c $\Rightarrow$ a<c

Он также должен быть нерефлексивным. Это означает, что объект не должен быть «впереди» самого себя.

a$\not<$a

Наконец, операция сравнения должна быть однонаправленной. Это означает, что объект не может находиться как до, так и после другого объекта. Более формально это записывается следующим образом:

a<b $\Rightarrow$ b$\not<$a

Все эти свойства справедливы для оператора сравнения, который устанавливает либо частичный, либо полный порядок. Для любой данной пары одно из них должно стоять перед другим. При полном порядке если два разных объекта не упорядочены в одну сторону, то они должны быть упорядочены по-другому:

a$\not<$b $\Rightarrow$ b<a (для разных a и b)

В случае частичного порядка дело обстоит иначе. Частичный порядок допускает как a$\not<$b, так и b$\not<$a.

Это относится не только к числам. Английские слова полностью упорядочены в алфавитном порядке, как показано в словаре. Частично они упорядочиваются с помощью оператора «содержит» (contains), так, например, «catalog» содержит «cat». Полный и частичный порядки можно найти для многих наборов. К ним относится набор исторических записей, или *фактов*.

## Предшественники

Способ, с помощью которого историческое моделирование приводит факты в частичный порядок, заключается в определении *предшественников*. Для каждого факта историческая модель делает явным, какие другие факты должны

были ему предшествовать. Это не просто список всех других фактов, которые имели место ранее во времени – в этом случае факты были бы расположены в последовательности – в полном порядке. Вместо этого предшественники – это факты, которые должны были произойти раньше, чтобы текущий факт имел смысл.

Если мы вернемся к нашему примеру с заказом на покупку, то увидим несколько предшественников. Заказ на покупку – это документ, подтверждающий решение покупателя приобрести товары у продавца. Пример заказа на покупку, приведенный ранее в этой главе, еще раз показан на рис. 2.7.


**Рис. 2.7.** Заказ на покупку

Заказ на покупку – это *факт*. Это историческая запись, которая документирует решение. Он неизменен – ни одна из сторон не может изменить сам заказ. Они могут только заменить его другим документом.

Факт заказа связан с несколькими другими фактами, которые появились раньше. Он относится к покупателю и продавцу как отдельным юридическим лицам. Эти субъекты были созданы со своим собственным набором исторических фактов-документов, которые были поданы в соответствующие регулирующие органы в качестве учредительных документов. Эти факты были созданы задолго до заказа на покупку.

Факт заказа на покупку также относится к товару. Историческим фактом является то, что виджет был опубликован в каталоге доступных товаров с указанной ценой. Каталог, однажды опубликованный, не изменяется. Он изменяется только путем публикации последующих каталогов, добавляющих и удаляющих продукты и изменяющих их цены. Связь заказа на покупку со всеми его предшественниками показана на рис. 2.8.

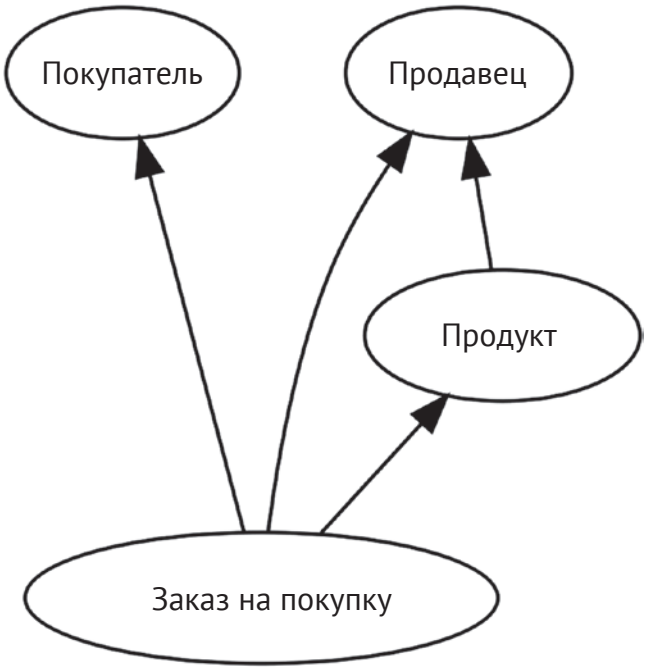


**Рис. 2.8.** Покупатель, продавец и продукт являются предшественниками заказа на покупку

В этой модели между заказами на поставку нет отношений предшественника. В ней не фиксируется, что один заказ на поставку был подан раньше по времени, чем другой. Предшественники – это не просто факты, которые произошли раньше во времени, это предпосылки – то, что должно быть истинным, чтобы данный факт имел смысл.

## Преемники

Полезно поговорить о направлении, противоположном отношению предшественников. Факт, который ссылается на другой факт, является его *преемником*. Преемники помогают нам развивать наше понимание системы с течением времени. Мы не можем изменить исторический факт, но можем создать преемников.

Продолжим рассказ о покупателе и продавце. Продавец получает копию заказа на покупку, а затем отправляет покупателю счет-фактуру. Счет-фактура – это еще один исторический факт. Предшественником этого факта является заказ на покупку. Преемником заказа на покупку является счет-фактура, как показано на рис. 2.9.

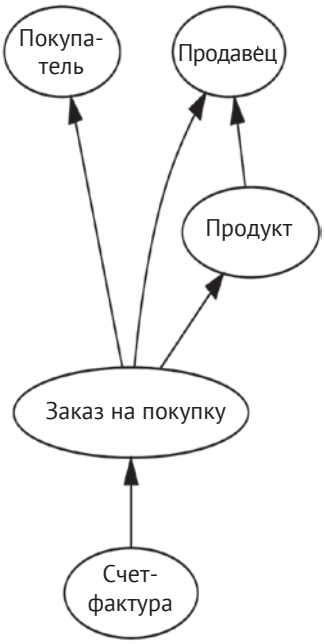


**Рис. 2.9.** Счет-фактура является преемником заказа на покупку

Наличие преемника не изменяет предшественника. Выставление счета-фактуры не изменяет историческую запись, которой является заказ на поставку. Однако преемник изменяет нашу *интерпретацию* предшественника. Когда мы видим счет-фактуру, мы знаем, что состояние заказа на покупку изменилось. Мы знаем, что счет был выставлен и было бы неправильно выставлять второй.

Важно понимать, что в исторической модели не существует механизма для *предотвращения* создания дополнительных преемников. Сама модель должна допускать несколько счетов-фактур к одному и тому же заказу на поставку. Если мы тщательно контролируем, *кто* может создавать эти счета-фактуры и на какой машине, то сможем избежать этой ситуации в любом практическом сценарии. Но сама модель не имеет возможности заблокировать заказ на покупку или предпочесть один счет-фактуру другому.

Факт не знает о своих преемниках. Со временем появляются новые преемники. Чтобы полностью понять состояние факта, мы должны запросить историческую модель, дабы узнать, были ли созданы новые преемники. Текущее состояние – это не проекция исторических фактов на изменяемые объекты, это просто коллекция известных преемников.

## Неизменяемые графы

Как и событие, исторический факт является неизменным. Но, в отличие от события, факт связан со своими предшественниками. Вместе взятые, эти свойства имеют интересные последствия.

Предшественники, на которых ссылается факт, сами являются неизменными фактами. Эти факты, в свою очередь, могут иметь предшественников. Это создает структуру, известную как направленный граф. Каждая вершина в этой структуре – это факт, а каждое ребро – это отношение предшественника. Эти отношения имеют направление: они направлены от преемника к предшественнику. Мы видели примеры таких графов, представленных ранее, когда они относились к заказам на покупку и счетам-фактурам. Еще один пример представлен на рис. 2.10.


**Рис. 2.10.** Факт ссылается на своих предшественников, которые, в свою очередь, ссылаются на своих предшественников

Поскольку факт ссылается на своих предшественников, а факт неизменяем, из этого следует, что предшественник не может быть добавлен к существующему факту. Это отношение предшественника является *частью* факта, и факт не может быть изменен. И поэтому, хотя можно добавить преемников к факту, невозможно добавить предшественников. Это соответствует нашему использованию отношения предшественника для определения того, что идет раньше в частичном порядке. Все предшественники должны быть известными фактами, записанными до нового факта.

От любого данного факта мы можем проследить граф по путям предшественников. Мы выберем подграф, рекурсивно включающий начальный факт, всех его предшественников и всех *их* предшественников. В результате этого процесса получается *транзитивное замыкание* начального факта. Если мы вычислим транзитивное замыкание реакции, то в итоге получим подграф, изображенный на рис. 2.11.

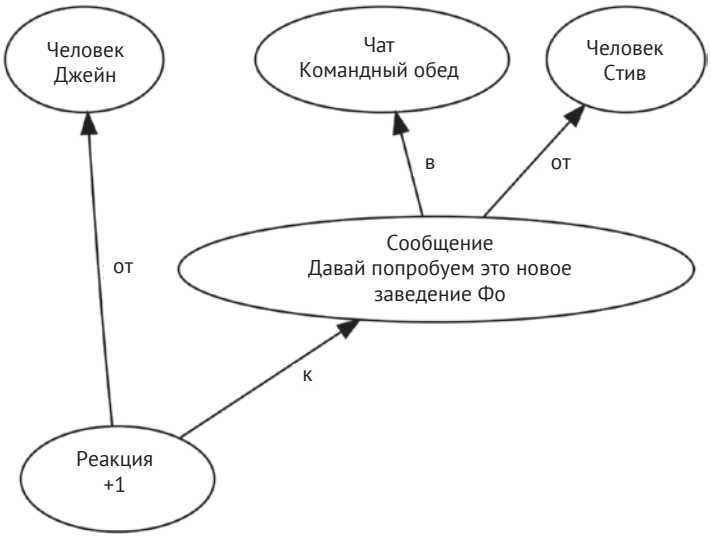


**Рис. 2.11.** Транзитивное замыкание факта содержит предшественников каждого факта

Чтобы построить транзитивное замыкание, мы начали с одного неизменного факта и пошли по стрелкам только в том направлении, которое не может измениться. Таким образом, подграф является неизменяемым. Для любого данного факта транзитивное замыкание всегда будет одним и тем же. Добавление новых преемников к любому из фактов в графе не изменит его. Эти преемники никогда не будут добавлены в транзитивное замыкание.

И наоборот, транзитивное замыкание *определяет* начальный факт. Не существует другого факта, для которого транзитивное замыкание дало бы тот же набор фактов. В исторической модели это единственный способ идентифицировать факт. Не существует глобальных уникальных идентификаторов (globally unique identifiers – GUID) или порядковых номеров вне этой структуры. Содержимое фактов в транзитивном замыкании – это все, что у вас есть, чтобы отличить один факт от другого.

## Совместная работа

Компьютеры в распределенной системе могут взаимодействовать, обмениваясь графами исторических фактов. При этом они должны быть уверены, что отправляют транзитивное замыкание каждого факта. Они должны знать, что получателю известны все предшественники на каждом шаге.

Когда компьютер записывает новую порцию информации – решение, принятое пользователем, или результат какого-либо бизнес-процесса, – он делает это, создавая новый факт. Но он не может создать этот факт на основе предшественников, о которых еще не знает. Он должен либо создать этих предшественников, либо узнать о них от своих коллег.

Отношения предшественников между фактами отражают структуру коммуникации между компьютерами. Преемник с одного компьютера может рассматриваться как ответ на своего предшественника, созданного на другом

компьютере. Когда вы наблюдаете отношение предшественник/преемник, это служит доказательством того, что два компьютера общались, чтобы обеспечить это. И наоборот, когда два факта не связаны между собой, то эти два факта могли быть созданы одновременно. Это и есть частичный порядок исторических фактов в распределенных системах. Неоднозначность упорядочения между несвязанными фактами делает компьютеры менее ограниченными и, как мы увидим, более способными действовать автономно.

## Ациклические графы

Неизменность фактов ограничивает их знанием своих предшественников на момент создания. Но есть еще два ограничения, которые мы должны наложить на систему. Во-первых, мы должны иметь возможность строить граф по одному факту за раз. Во-вторых, мы не можем позволить факту ссылаться на себя как на предшественника. Мы должны запретить как одновременное создание, так и ссылку на себя, чтобы не создавать циклов.

Каждый граф начинается пустым. Он не содержит никаких фактов. Поэтому первый факт, добавленный в граф, не может иметь предшественников. Нет никаких данных, на которых можно было бы построить граф. Первый факт – это корень. Граф, содержащий только один корень, не имеет циклов, потому что в нем нет ребер.

Пусть пройдет время, и в граф будут добавлены новые факты. Предположим, что граф по-прежнему не содержит циклов. Когда я добавляю новый факт в граф, этот факт может ссылаться на любой из существующих фактов в качестве предшественников. Однако существующие факты не могут ссылаться на новый факт как на предшественника. Я не могу изменить их отношения предшественников, а этот новый факт еще не существует. Поэтому я не могу ввести цикл, добавив один факт.

Если бы мы разрешили ссылку на себя, то могли бы ввести тривиальный цикл. А если бы мы разрешили одновременную вставку, то могли бы ввести два факта, которые имеют друг друга в качестве предшественников. Поскольку ни одна из этих операций не разрешена, результирующий граф фактов не должен содержать циклов. В математике такая структура известна как направленный ациклический граф и обладает многими интересными свойствами. По мере того как мы будем углубляться в аналитические и реализационные детали исторического моделирования, мы будем использовать все преимущества ациклической природы графа.

## Своевременность

В системе, основанной на обмене историческими фактами, не все стороны будут знать обо всех фактах в одно и то же время. Это одно из самых сильных мест исторического моделирования, но также и одно из его важных ограничений. Невозможно отвергнуть факт, основываясь на времени, когда вы узнали о нем. Причина в том, что другие стороны могли узнать о нем раньше и, следовательно, пришли бы к другому выводу относительно этого факта. Чтобы каждая сторона в системе в конечном счете пришла к одному и тому же выводу, этот вывод не может быть основан на своевременности.

Это вызывает значительные проблемы в системах, которые не признают данное ограничение. Некоторые юридические документы, такие как налого-

вые формы, чеки и счета-фактуры, имеют четкие сроки исполнения или истечения срока действия. Если форма получена после требуемой даты, она не будет принята к исполнению. Отправитель должен приложить максимум усилий, чтобы доказать, что документ был написан и передан вовремя, или же страдать от последствий неудачной транзакции. В таких ситуациях отправитель считает одно – что он уложился в срок, а получатель верит в другое. Только арбитраж центрального органа может разрешить эти ситуации.

Чтобы разработать систему, не зависящую от центрального органа, мы должны признать, что документы будут получены с опозданием. В истинно исторической модели факт не отвергается на основании времени, когда он был получен. В лучшем случае мы можем зафиксировать тот факт, что был наложен штраф или возможность была упущена из-за того, что информация не поступила в определенное место к определенному времени. Но мы не можем доказать, что информация не существовала в другом месте в это время. И когда этот факт появляется позже, мы должны решить, как нам реагировать на него. Все стороны должны признать существование фактов, независимо от того, когда они о них узнали, и сделать один и тот же вывод. Возможно, этот вывод заключается в том, что отправитель все еще должен заплатить штраф. Но своевременность сама по себе не определила такой исход.

Таковы правила исторической модели. Они логически вытекают из желания отразить всю историю системы с несколькими сторонами, разделенными временем и пространством, и при этом обменивающимися историческими фактами. Эти факты должны быть неизменными. Два факта, имеющих одинаковое транзитивное замыкание, действительно являются одним и тем же фактом. Мы не можем гарантировать и поэтому не можем полагаться на то, что у любого данного факта есть только один преемник. И мы не можем изменить нашу интерпретацию истории, основываясь на своевременности нашего знания о ней.

## Ограничения исторического моделирования

В оставшейся части книги мы сосредоточим свое внимание на историческом моделировании. Другие формы неизменяемой архитектуры описаны в других местах, но историческое моделирование требует немного большего изучения. Этот выбор, как мы увидим, предлагает множество преимуществ – автономность, расширяемость и разрешение конфликтов, и это лишь некоторые из них. Но он не лишен ограничений. Мы уже упоминали о некоторых из них, но теперь давайте рассмотрим их более подробно.

С возможностями исторического моделирования связаны некоторые ограничения. Эти ограничения делают неуместным применение исторического моделирования к определенным типам систем. В таких ситуациях лучше всего моделировать всю систему или ее часть статически – то есть используя метод, который фиксирует текущее состояние, и интегрировать там, где это необходимо. К счастью, доступны хорошие стратегии интеграции.

Мы часто обнаруживаем, что можем объединить историческую модель со *статической* моделью. Статическая модель, как следует из ее названия, основана на состоянии. Модель является изменяемой, централизованной

и может обеспечивать серийный доступ. Реляционные базы данных являются хорошими статическими моделями, поскольку они хорошо поддерживают эффективную блокировку.

## Отсутствие центральной власти

Историческая модель позволяет принимать решения автономно. Каждое решение записывается в локальную историю и в конечном итоге передается остальной части системы. В результате система не может отвергать факты на основании возраста или текущего состояния.

Решения, которые были приняты в прошлом, утверждаются локально, с использованием только той информации, которая доступна в текущий момент времени. Нет необходимости консультироваться с удаленной частью системы. Это решение не может быть отвергнуто постфактум.

Такой подход делает историческое моделирование неприемлемым для тех частей системы, где требуется центральный орган. Например, система бронирования конференц-залов должна точно знать, был ли свободен тот или иной зал в определенное время. Когда бронирование утверждается, утверждающий должен знать, что никакая другая бронь на тот же зал в то же время не была утверждена. Это решение должно приниматься центральным органом.

Историческая модель может применяться центральным органом, если только сам центральный орган использует статическую модель. Историческая модель может фиксировать тот факт, что был сделан запрос. Это происходит в точке запроса, например на рабочем месте пользователя или на устройстве, установленном у двери, и эти факты попадают в центральный орган. Историческая модель также может фиксировать факт одобрения запроса. Это происходит в центральном органе и передается на устройства. Но историческая модель сама по себе не может точно сказать, свободна ли комната в любой момент времени. Для этого необходимо, чтобы модель знала, что бронирование не было одобрено, что невозможно.

Чтобы решить эту проблему, система должна включать центральный орган со статической моделью. Историческая модель регистрирует запросы на резервирование и утверждения, а статическая модель определяет доступность. Центральный орган не обязательно должен быть одной машиной, это может быть кластер машин. До тех пор, пока члены этого кластера имеют доступ к одной и той же статической модели, они могут действовать с единой властью. Для общей статической модели необходим механизм блокировки, чтобы помочь этому кластеру координировать свои действия. Реляционные базы данных поддерживают транзакции, которые хорошо отвечают этой потребности.

Центральный орган будет записывать одобрение или отклонение запроса как последовательный факт. Этот факт будет возвращаться к клиенту, от которого поступил запрос. Историческая модель обеспечивает все преимущества, о которых говорилось ранее, – полную историю запроса, в конечном итоге согласованное представление текущего состояния и объединяемый механизм связи. Единственный компонент, который лучше моделировать статически, – это тот, который требует центрального управления: наличие свободных залов.

## Отсутствие часов реального времени

Запрос, чувствительный ко времени, должен быть выполнен в течение определенного периода времени. Если это не так, запрос считается недействительным. Подобные запросы часто встречаются в системах реального времени, таких как заводская автоматизация. Запрос на открытие двери или перемещение роботизированного манипулятора должен быть выполнен в течение короткого промежутка времени. Если сообщение не приходит вовремя, то запрос должен быть отклонен.

Факты в исторической модели, однако, выполняются независимо от временных рамок. Решение принимается в момент регистрации факта и не может быть отклонено после этого. Для передачи факта может потребоваться неопределенный период времени. Получатель просто информируется о чем-то, что уже произошло в истории.

Хотя может быть целесообразно моделировать исторически вход или выход заводской системы автоматизации, программное обеспечение, которое управляет самим предприятием, должно использовать модель реального времени. Такие модели специально разработаны для обеспечения чувствительного ко времени поведения в режиме отказоустойчивости. Если сообщение не приходит в нужное время, система по умолчанию переходит в безопасный режим работы. А когда сообщение все-таки приходит, пусть и с опозданием, система игнорирует его, чтобы не причинить вреда.

## Отсутствие ограничений уникальности

В исторической модели любой запрос на преемников факта может вернуть несколько результатов. Невозможно ограничить запрос, чтобы он возвращал только один результат. Следствием является то, что область, требующая не более одного результата, не может быть эффективно смоделирована исторически.

Например, вход в систему, требующий уникального имени пользователя, должен поддерживаться статической моделью. Историческая модель не сможет обеспечить уникальность имени пользователя.

В лучшем случае историческая модель может включать факт, содержащий только имя пользователя. Поскольку факт уникально идентифицируется по его значению, логически существует только один факт с таким точным именем пользователя. Однако факт не может содержать ничего, что могло бы отличаться от одного пользователя к другому. Если бы он содержал информацию в дополнение к имени пользователя, то два или более фактов могли бы существовать с одним и тем же именем. Они больше не будут уникальными.

Поэтому, чтобы соотнести отдельное имя пользователя с пользователем, потребуется факт-преемник. Для идентификации пользователя по данному имени пользователя потребуется запрос на преемников факта имени пользователя. Такой запрос не может гарантированно вернуть не более одного факта. Всегда существует возможность того, что он вернет больше.

Для моделирования системы, требующей ограничений уникальности, необходимо использовать статическую модель. К модели можно обратиться, чтобы определить, используется ли уже нужное значение. Индексирование и транзакционные возможности реляционной базы данных снова вступают в игру.

Эта статическая модель также должна быть расположена централизованно. Реплика статической модели не может обеспечить уникальность. Вставка в одну копию должна быть заблокирована, чтобы обратиться к другим. Только если это уникальное значение не зарезервировано в кворуме (обычно простом большинстве) копий, оно может быть принято. Алгоритм консенсуса, такой как Paxos[1], может быть использован для достижения кворума.

Если требуется уникальность, например при регистрации имени пользователя, историческая модель может быть использована для запросов на регистрацию, а также для ответов о ее принятии или отказе. Запросы могут быть записаны, как факты, клиентами системы. Эти факты попадают в центральный орган, который имеет доступ к статической модели. Статическая модель обеспечивает ограничения уникальности. Центральный орган решает, следует ли одобрить или отклонить запрос на основе статической модели, а затем записывает это решение как факт-преемник.

Ответ вернется к клиенту, от которого пришел запрос, и только тогда клиент узнает, является ли запрошенное имя пользователя уникальным. Он сформирует запрос для получения преемников факта запроса – принятия или отклонения. Как только у него появится один преемник, он узнает ответ на вопрос об уникальности. Однако в самой исторической модели нет гарантии, что у запроса не будет преемников. Такая гарантия возникает только благодаря доверию к центральному органу, принимающему решение с помощью статической модели, которая может обеспечить уникальность.

## Отсутствие агрегирования

Можно ожидать, что после определенного времени активности система предоставит совокупность или резюме деятельности за этот период. Например, финансовая книга может быть закрыта в конце дня, месяца или квартала. Затем система выдает сводку, в которой записана общая сумма операций за этот период. С этого момента никакие дополнительные транзакции не допускаются.

Историческая модель не может гарантировать, что все факты за определенный период были замечены. Система, ответственная за генерацию совокупности, может не иметь всех записей за период в требуемое время. Если она получает факт после вычисления и записи сводки, то по правилам исторического моделирования она не имеет права отвергать его. Решение было принято в другом месте, и факт этого решения был просто передан.

Для решения проблемы агрегирования исторических фактов существуют три стратегии: централизованный реестр, map-reduce и блокчейн. Централизованный реестр, безусловно, является самой простой стратегией. Она основана на статической модели подсчета того, какие факты были включены в какой период. Например, она определяет, какие финансовые операции относятся к той или иной дате бизнеса или квартальной сводки. Она принимает это решение в рамках подсчета по мере поступления фактов, независимо от того, когда они

[1] Паксос (англ. Paxos) — семейство протоколов для решения задачи консенсуса в сети ненадежных вычислителей. Консенсус — процесс получения согласованного результата группой участников, основная проблема — наличие помех в среде передачи данных. – *Прим. перев*.

произошли в истории. Подсчет – это статическая модель. Центральный орган использует эту статическую модель, чтобы гарантировать, что транзакция не будет дважды учтена, другими словами, включена в более чем один период.

Map-reduce децентрализует статическую модель. Больше нет необходимости в том, чтобы одна статическая модель содержала все финансовые транзакции, которые происходят в течение рабочего дня. Вместо этого транзакции распределяются между несколькими статическими моделями, называемыми шардами. Чтобы вычислить совокупность за дату или квартал, координатор посылает запрос в каждый из шардов. Каждый шард вычисляет свой собственный агрегат, а затем передает результат координатору. Координатор объединяет все агрегаты в один окончательный ответ. Это работает, потому что ни одна транзакция никогда не дублируется между шардами. Иначе это дублирование привело бы к завышению окончательного результата.

Блокчейн более сложен, но позволяет избежать необходимости в центральном органе. Во многих точках системы отдельные факты собираются в блоки-кандидаты. Хеш каждого блока вычисляется и проверяется на некоторое произвольное условие (например, определенное количество ведущих нулей). Это произвольное условие является подтверждением работы, которое гарантирует, что удовлетворительные блоки будут найдены на желаемой частоте. Каждый блок-кандидат содержит хеш своего предшественника, и ни один факт не может встречаться более, чем в одном блоке в цепочке. Узлы внутри системы позволяют выбрать самую длинную цепочку удовлетворительных блоков.

При проектировании системы, требующей агрегирования по истории, добавьте статическую модель – будь то единичная или чередующаяся – к исторической. Моделируйте отдельные транзакции исторически. Центральный орган собирает список текущих исторических фактов в статическую модель. Через регулярные промежутки времени завершайте подсчет фактов и вычисляйте итог – либо в виде агрегатной функции, либо с помощью map-reduce. Это сохраняет логические и технические преимущества исторического моделирования, позволяя при этом агрегировать данные.

# Глава 3

# Как читать историческую модель

На протяжении всей книги мы будем изучать множество примеров исторических моделей. Для этого нам понадобится язык для их описания. Этот язык будет частично визуальным и частично текстовым. Визуальный аспект этого языка будет способствовать общему пониманию, а текстовая часть обеспечит конкретизацию.

Целью языков моделирования, визуальных или текстовых, является достижение общего понимания решений, которые мы принимаем коллективно, и последствий этих решений. Бизнес-аналитики, владельцы продуктов и информационные архитекторы будут открывать язык и правила домена. Разработчики и системные архитекторы опишут последствия различных решений. А дизайнеры пользовательского опыта отобразят модель в интерфейсы, управляемые задачами.

На протяжении всего процесса команда общается на общем языке. Этот язык должен быть максимально свободен от жаргона и деталей реализации. Он не должен говорить о базах данных, API, сервисах или хранилищах. Вместо этого он должен фокусироваться на сущностях и действиях в проблемной области.

При этом язык должен быть точным. Не должно быть места для двусмысленности в требованиях к системе. Когда разработчики реализуют решение в соответствии со спецификацией, не должно быть никакой путаницы относительно того, как будет вести себя корректная программа. Более того, спецификация должна быть даже достаточной для генерации некоторых частей кода.

Начиная с базового уровня неизменяемой архитектуры, мы можем определить именно такой язык спецификации. Он несет в себе достаточно ограничений, чтобы гарантировать, что его корректная реализация будет иметь желаемые характеристики, такие как производительность, масштабируемость, безопасность и автономность. И поскольку одним из этих ограничений является неизменяемость, будут возможны рассуждения о модели.

Язык моделирования, который мы будем описывать, – историческое моделирование – имеет два графических компонента и один текстовый компонент. Мы начнем с первого из графических компонентов – графа типов фактов.

## Графы типов фактов

В рамках визуального языка мы будем создавать два вида графов. Один будет представлять типы фактов, а другой – экземпляры. Графы типов фактов будут более распространенными из них, поэтому давайте сначала опишем их.

В графе типа факта тип факта представлен в виде помеченного эллипса, как показано на рис. 3.1.


**Рис. 3.1.** Один тип факта

Стрелка между двумя типами фактов указывает на отношение предшественника и преемника. Тип в начале стрелки является предшественником того, который находится в конце. На рис. 3.2 показан пример.

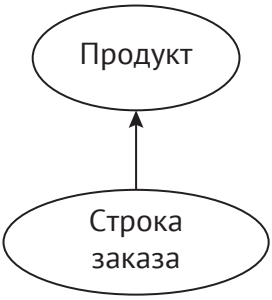


**Рис. 3.2.** Стрелки направлены к предшественникам

В любом отношении «предшественник/преемник» предшественник может иметь ноль или более преемников. В исторической модели нет возможности ограничить число преемников для любого факта. Даже если на графе показана только одна строка заказа, это означает, что у продукта может быть много строк заказа.

Однако сторона отношения «предшественник» может быть ограничена. На предыдущем графе строка заказа связана ровно с одним продуктом. С помощью дополнительных обозначений мы можем указать другие значения. На рис. 3.3 ноль или один предшественник обозначен вопросительным знаком.

В этом графе строка счета-фактуры может ссылаться или не ссылаться на строку заказа. Ссылка является необязательной. Некоторые строки счета-фактуры представляют собой плату за конкретные заказанные продукты и поэтому имеют строку заказа. Другие – это сборы или скидки, не связанные с конкретной строкой заказа. Они не имеют предшественника.

Третья кардинальность, которая может быть изображена на графе типов фактов, – это ноль или более (т. е. много) предшественников. Чтобы указать это значение, мы используем звездочку, как показано на рис. 3.4.

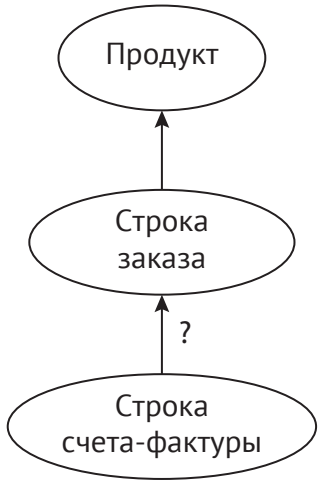


**Рис. 3.3.** Вопросительный знак указывает на необязательного предшественника

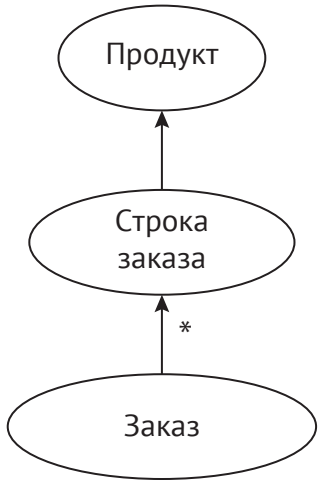


**Рис. 3.4.** Звездочка указывает на наличие нескольких предшественников

Этот граф говорит о том, что заказ относится к нулю или более строкам заказа. Поскольку хвост стрелки допускает увеличение количества, это отношение «многие ко многим». В этом графе присутствует не только число элементов. Он также представляет порядок, в котором факты могут быть созданы, и степень, в которой изменения могут быть применены. Стрелка на рис. 3.4 допускает несколько строк заказа в качестве предшественников. Стрелка на рис. 3.5 повернута и не имеет звездочки. Тем не менее она по-прежнему допускает несколько строк заказа, только теперь в качестве преемников.

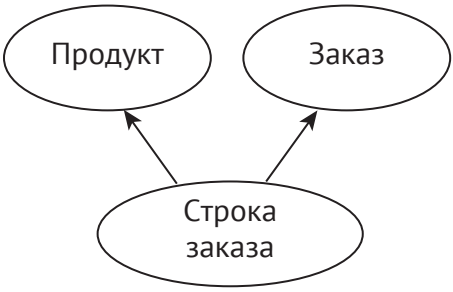


**Рис. 3.5.** Отношения преемственности всегда допускают множественность, поэтому данная модель по-прежнему допускает несколько строк заказа

На рис. 3.5 направление отношения «предшественник/преемник» между заказом и строкой заказа изменено на противоположное. Преемник всегда допускает ноль или более фактов, и поэтому оба графа указывают на множество строк заказа к заказу. Различие, однако, связано с неизменяемостью. На рис. 3.4 множество строк заказа являются предшественниками заказа. Они были известны при создании заказа. Новые строки заказа не могут быть добавлены к заказу после его создания. Существующие строки заказа не могут быть удалены после создания заказа.

Коллекция строк заказа неизменна. Когда мы хотим указать на эту неизменяемость, мы часто говорим, что преемник захватывает своих предшественников, как «заказ захватывает множество строк заказа».

В данной области мы хотим зафиксировать строки заказа в заказе и поэтому сделаем строку заказа предшественником (как на рис. 3.4). Однако мы введем концепцию корзины, в которую клиент может добавлять строки заказа. Важно выбирать модель на основе направления отношений «предшественник/преемник», а не только на основе их количества.

Для некоторых моделей, которые мы изучим в ближайшее время, имеет смысл, чтобы факт ссылался на другие факты своего типа. Мы обозначаем это как цикл, как показано на рис. 3.6.

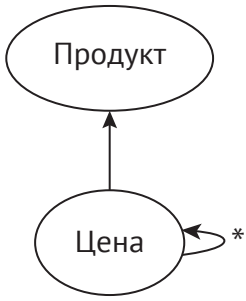


**Рис. 3.6.** Цикл указывает на то, что предыдущие цены являются предшественниками

Цена относится не только к продукту как предшественнику, но и к нулю или нескольким ценам, которые были до этого. Это указывает на то, что цена заменяет предыдущие значения. Несмотря на то что это отношение вводит цикл

в граф типов фактов, оно не допускает циклов в фактах экземпляров. Данный экземпляр цены не может ссылаться на себя как на предшественника. Он может только ссылаться на предыдущую цену, ту, которую он заменяет.

Должна была существовать первая цена на продукт. Этот экземпляр факта не будет иметь предшественников цены. Вот почему мы не можем представить это как единственное и неповторимое отношение. Циклы в графе типа факта обязательно включают стрелку, которая является либо необязательной (ноль или один), либо, что более распространено, множественной (ноль или более). Поскольку эти ссылки являются единственным способом моделирования свойств, которые изменяются со временем, мы часто называем их «изменяемыми», как, например, «товар имеет изменяемую цену».

Если собрать все это вместе, то полная модель для домена заказов выглядит так, как показано на рис. 3.7.


**Рис. 3.7.** Граф типов фактов, показывающий различные виды обозначений

Продукт в каталоге имеет изменяемую цену. Строка заказа в корзине фиксирует конкретную цену для продукта. В заказе фиксируется набор строк заказа. Строки заказа могут быть добавлены в корзину, но заказ закрывается. Строка счета-фактуры может ссылаться на строку заказа, а может и не ссылаться на нее, а счет-фактура фиксирует набор строк счета-фактуры.

Диаграмма типов фактов выражает как кардинальность, так и причинность области. При ее прочтении открывается рассказ о том, как система оказалась в определенном состоянии, и раскрываются ограничения на то, как эта система может и не может измениться.

## Шахматная партия

Когда я впервые задумался об историческом моделировании, я нарисовал несколько моделей, чтобы убедиться в том, что это будет полезный способ анализа программного обеспечения. Первая модель, которую я нарисовал, была моделью одной шахматной партии. Шахматы – это игра между двумя игроками. Игроки выбираются до начала игры. Поэтому игроки являются предшественниками игры, как показано на рис. 3.8.

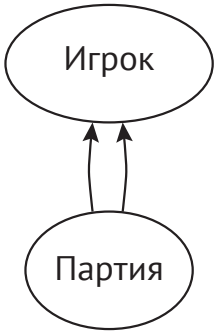


**Рис. 3.8.** В шахматной партии участвуют два игрока

Это сразу же показало, что мне нужен способ говорить о различных предшественниках одного типа. Сначала я подумал о том, чтобы разрешить массив игроков и использовать индекс для указания порядка игры. Углубившись в модели, я понял, что предшественники должны быть множествами, а не массивами, что отменяет эту идею. Кроме того, игра поддерживает ровно двух игроков. Массив позволил бы представить недопустимые игры с 0, 1 или 3+ игроками. Естественным выводом было то, что отношения предшественников должны поддерживать метки, как показано на рис. 3.9.

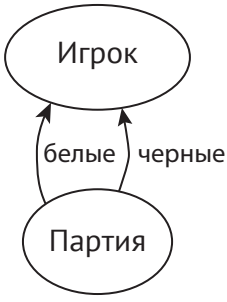


**Рис. 3.9.** Игроки в шахматной партии имеют метки

Когда не требуется однозначного определения, метки можно не указывать. Метка всегда описывает предшественника, а не преемника.

## Важные атрибуты

В шахматной литературе место, где играется партия, часто используется в названии. Оно представляет собой важный атрибут модели. Моим первым побуждением было записать этот атрибут в факте партии. Затем можно было бы отобразить эту информацию как часть названия партии.

Однако при дальнейшем размышлении становится ясно, что это было бы не совсем правильно. Несколько партий будут сыграны в одном и том же месте. Если место – это просто поле игры, то модель не совсем верно отражает эту истину. Название места – это не само место. И нет способа найти все партии, сыгранные в определенном месте.

Решение, как показано на рис. 3.10, состоит в том, чтобы представить место как факт. Место является предшественником партии, оно существовало до того, как была сыграна партия, и партия знает об этом месте. Партию нельзя переместить в другое место. Место является неизменным и является частью идентичности партии.

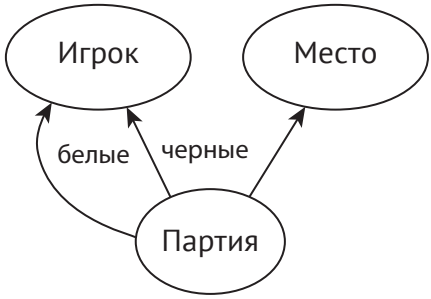


**Рис. 3.10.** Место отделено от партии и стало ее фактом

Благодаря этому изменению модель точно отражает важность места. Это факт сам по себе. Он стоит отдельно от любой партии, но также дает возможность найти набор партий. Если важный атрибут скрыт в виде поля в другом факте, то у нас нет возможности говорить о таком запросе. Но если важный атрибут представлен в виде отдельного факта, то мы можем рассуждать о нем соответствующим образом.

## Цепочка фактов

Следующей частью партии, которую мне нужно было смоделировать, были ходы. Учитывая, что предшественник – это вещь, которая произошла раньше, я решил представить порядок ходов с помощью отношений предшественников. В результате получилась диаграмма, показанная на рис. 3.11.

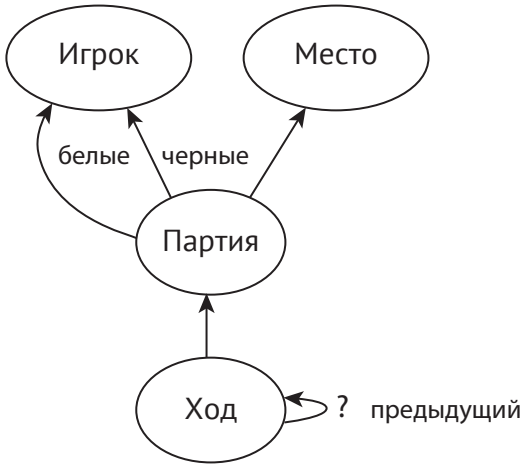


**Рис. 3.11.** Ход ссылается на предыдущий ход как на своего предшественника

Вопросительный знак указывает на то, что предыдущий ход необязателен, первый ход не будет его иметь. Предшественник фиксирует только непосредственно предыдущий ход, так как предыдущий ход допускает *необязательных* предшественников, а не *множество* предшественников. Вся история партии может быть найдена в цепочке предшественников.

Хотя модель требует, чтобы только один ход *предшествовал* другому ходу, она не требует, чтобы только один ход *следовал* за другим. Структура модели не препятствует игроку схитрить, сыграв два возможных хода.

Структура также не препятствует тому, чтобы ход следовал за ходом из другой партии. Нет никаких ограничений на то, что предыдущий ход принадлежит той же партии. Это должно быть выражено в правиле проверки.

После анализа этой модели я не был особенно доволен отношениями предшествования. Я понял, что последующие ходы в игре будут определяться длинным списком предыдущих ходов. Это казалось расточительным. Я не мог дать количественную оценку этой проблеме, потому что еще не определился с тем, как реализовать эту модель. Но я решил, что ход может просто содержать номер хода, а также затронутые клетки. Начиная с нуля, белые будут делать четные ходы, а черные – нечетные. Это позволило сгладить модель и сделать ее такой, как показано на рис. 3.12.

Модель больше не представляет структуру партии. Она не фиксирует идею о том, что один ход следует за другим. Но опять же, учитывая невозможность запретить вилки, структура не представляла особой ценности. Более того, удаление предыдущей ссылки устраняло возможность пересечения ходов в партиях. Моделирование – это компромисс между строгостью системы типов и грубой силой проверки. Проверка последовательного порядка ходов делает модель более простой.

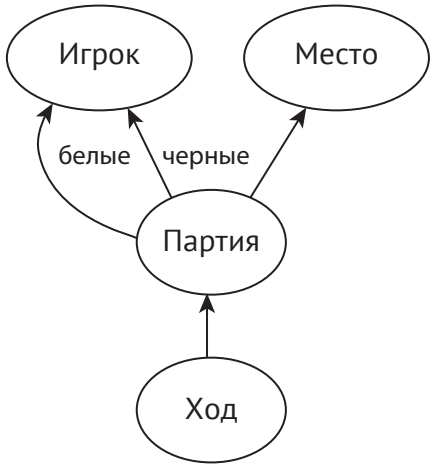


**Рис. 3.12.** Из модели удалены ссылки на предыдущие ходы

## Исход партии

Последней частью партии, которую необходимо смоделировать, был результат. Была ли это победа белых (1:0), победа черных (0:1) или ничья (½:½)? Самое простое решение – создать факт результата и записать это значение в поле, как показано на рис. 3.13.

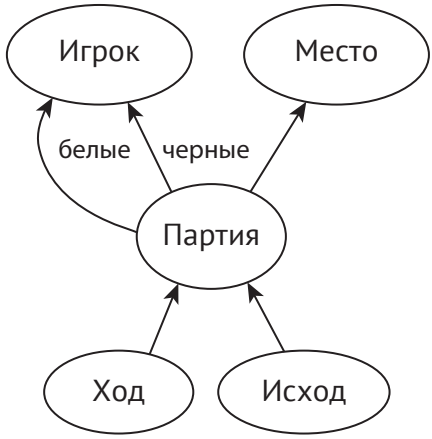


**Рис. 3.13.** Новый факт фиксирует исход партии

Я обнаружил две проблемы с этим простым решением. Во-первых, оно не предлагает никакой помощи при поиске партий, которые выиграл определенный игрок. Это может быть важным запросом, поэтому модель должна его поддерживать. Во-вторых, оно позволяло продолжать ходы после окончания партии. Исход должен был блокировать дальнейшие ходы.

Чтобы решить первую проблему, я решил разделить факт исхода на два разных типа – победа и ничья. Победа относится к одному из двух игроков.

Ничья – чтобы помочь с запросом – относится к обоим. Это меняет модель на показанную на рис. 3.14.

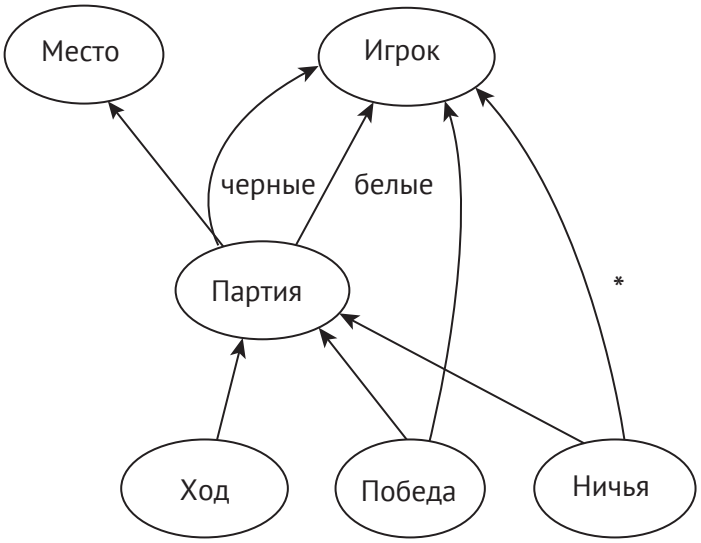


**Рис. 3.14.** Партия завершается победой для одного игрока или ничьей для обоих

Модель снова требует проверки, чтобы убедиться, что победителем должен быть один из игроков. Но, с другой стороны, гораздо проще запросить все выигрыши и ничьи игрока. Это может помочь в таких вещах, как подсчет очков.

Чтобы решить вторую проблему, я воспользовался тем фактом, что отношения предшественников неизменны. Если исход записывает все ходы партии в качестве предшественников, то эти ходы фиксируются при завершении партии, как показано на рис. 3.15. Конечно, можно было бы записать и факты будущих ходов, но они не способствовали бы получению результата.

Как показывает этот пример, моделирование проблемы исторически раскрывает несколько решений, которые в противном случае могли бы быть отложены как детали реализации. Правильный вариант не всегда очевиден. Но выбор имеет важные последующие эффекты. Один путь может привести к более простой модели, но сложному запросу. Другой может ограничить систему, чтобы сделать недопустимые входные данные непредставимыми, что приведет к слишком большой нагрузке на систему типов. Потратьте время на разработку нескольких альтернатив и проанализируйте преимущества каждой из них.

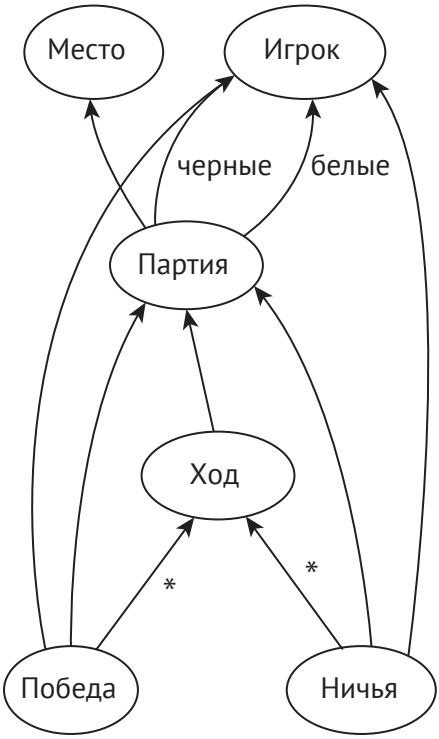


**Рис. 3.15.** Победа или ничья фиксирует набор ходов

## Графы экземпляров фактов

До сих пор графы, которые мы рисовали, относились к *типам* фактов. Визуальный язык исторического моделирования также включает в себя вид графа, который представляет *экземпляры* фактов. Этот вид графов мы будем использовать реже. Цель графа экземпляров фактов – проиллюстрировать конкретный пример состояния системы в определенный момент времени и на определенном узле. Он включает больше деталей о наблюдаемых экземплярах фактов, но обычно содержит только небольшое количество фактов.

Экземпляры фактов отличаются от типов фактов тем, что они рисуются без границы. Вместо этого тело факта образует прямоугольник текста с разделителем между типом и содержимым, как показано на рис. 3.16.

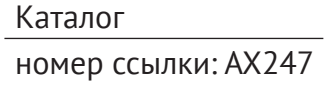


**Рис. 3.16.** Экземпляр каталога фактов показан со всеми его параметрами

Отношения предшественник/преемник рисуются стрелками, как на диаграмме типов, но на диаграмме экземпляров не указывается количество. Никакие отношения не включают звездочку или вопросительный знак. На рис. 3.17 показана связь между каталогом и одним из его продуктов.


**Рис. 3.17.** Продукт указывает на каталог своего предшественника

Если ссылка на предшественника имеет значение «необязательный» или «много», она допускает ноль предшественников. Экземпляр, не имеющий предшественников в этой роли, рисуется на диаграмме экземпляров как прерывающаяся линия. Если роль необязательна (?), то ограничитель представляет собой нулевую ссылку. Если роль является множественной (*), как в данном примере, то ограничитель представляет собой пустое множество.

Ограничители полезны для иллюстрации изменений в изменяемом свойстве. Корневой экземпляр или начальное значение изменяемого свойства не имеет предшественника. Последующие экземпляры образуют цепочку от корня, как показано на рис. 3.18.

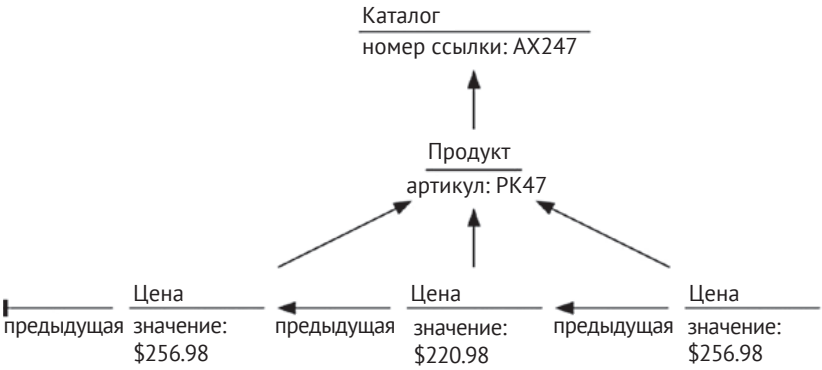


**Рис. 3.18.** Первая цена не имеет предшествующей

Вспомните, что на диаграмме типов фактов был изображен только один узел цены. Небольшая петля указывала на то, что у цены было ноль или более предыдущих цен. Теперь на диаграмме экземпляров фактов мы видим, что на самом деле означает эта петля. Она не указывает на то, что модель допускает циклы. Это означает только то, что экземпляры данного типа могут ссылаться на другие экземпляры того же типа. Цикл превратился в цепь.

Диаграмма экземпляров фактов не допускает циклов. Факт никогда не ссылается на себя как на предшественника. Факты также не могут ссылаться на предшественников, которые, в свою очередь, ссылаются на оригинал, прямо или косвенно.

Поскольку все экземпляры цены ссылаются на общий предшественник Продукт, для группировки общих отношений можно использовать сокращенное обозначение. Вокруг всех членов группы рисуется рамка, как показано на рис. 3.19. Ссылка на предшественника, общая для всех членов группы, показана в виде линии, исходящей из этой рамки. Если преемник относится ко всем членам группы, он будет представлен в виде стрелки, направленной внутрь.

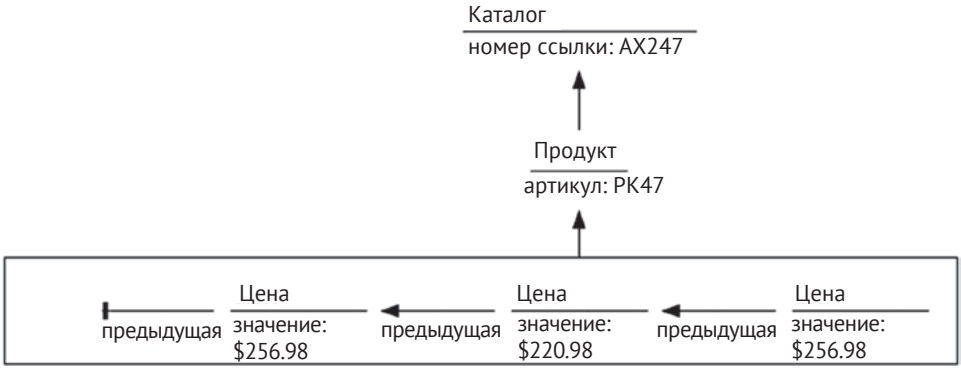


**Рис. 3.19.** Рамка указывает на то, что все экземпляры имеют общего предшественника

Если диаграммы типов фактов представляют собой общее описание модели, то диаграммы экземпляров фактов показывают конкретные примеры. Диаграммы экземпляров помогают проиллюстрировать последствия различных решений по моделированию. Они предлагают форму отладки до того, как модель была реализована. Давайте рассмотрим конкретный пример и покажем, как они помогают обосновывать проектные решения.

## Бессмертная партия

Ранее мы представляли, как может выглядеть любая шахматная партия. Теперь нарисуем конкретный пример шахматной партии, используя эту модель. Перед началом игры у нас есть два игрока – Андерсен и Кизерицкий. Они встречаются в Лондоне. Как показано на рис. 3.20, партия – это факт, который соединяет этих двух игроков в данном месте в определенный момент времени.

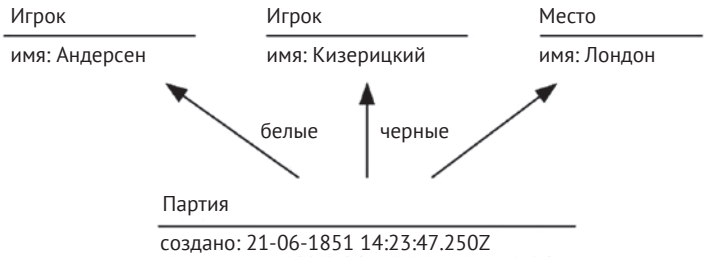


**Рис. 3.20.** Партия между Андерсеном и Кизерицким в Лондоне в 1851 году

Важно, что мы включили поле «создано» (createdAt) в партию. Оно представляет собой момент, когда, с точки зрения клиента, был создан факт. В данном примере мы используем точный момент начала матча в Лондоне. (Время здесь вымышленное, так как записей такого уровня точности не существует.)

Если бы мы моделировали игру без поля createdAt, то любая партия между Андерсеном белыми и Кизерицким черными, сыгранная в Лондоне, была бы той же самой партией. Факт однозначно идентифицируется по его типу, значениям его полей и набору его предшественников. Эта модель делает разумное предположение, что два игрока не будут одновременно начинать две разные партии.

Также важно, что createdAt представляет момент, в который факт стал фактом, а не момент, когда он стал известен какому-либо конкретному компьютеру. Конечно, в то время в Лондоне не было компьютеров, которые могли бы зафиксировать этот факт. Тем не менее он существовал. В современных системах время создания часто фиксируется в виде временной метки на компьютере клиента. После этого, однако, другие компьютеры, которые узнают об этом факте, должны соблюдать эту временную метку, независимо от того, сколько времени прошло.

## Регистрация ходов

Теперь, когда факт партии зафиксирован, игроки начинают делать ходы. Мы можем фиксировать эти ходы как факты, относящиеся к партии. На рис. 3.21 показана партия после первых трех ходов.

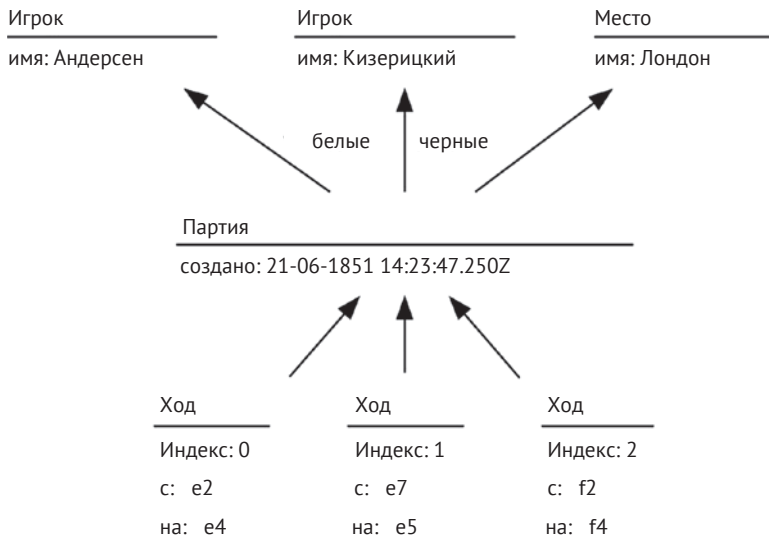


**Рис. 3.21.** Ходы фиксируются как преемники факта партии

В предыдущей итерации модели мы рассматривали возможность представления отношения между ходом и предыдущим ходом в качестве предшественника. Преимущество такого представления заключалось в точности – ход является предшественником того, который следует за ним. Но это имело более существенный недостаток – создание длинных цепочек. Если бы мы выбрали эту модель, то партия после трех ходов выглядела бы как на рис. 3.22.

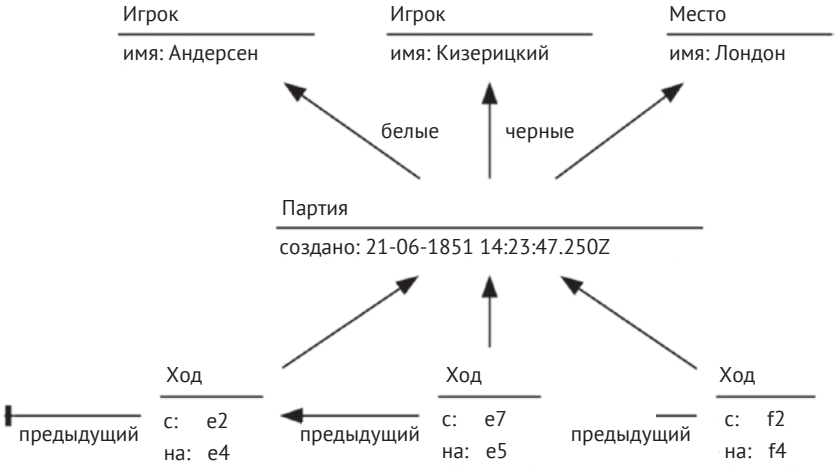


**Рис. 3.22.** В альтернативной модели ходы представлены как предшественники друг друга

По мере продолжения партии цепочка будет становиться все длиннее. Понимание того, что 20-й ход является 20-м ходом, требует прохождения всей этой цепочки. Даже *определение* 20-го хода означает разговор обо всей истории, поскольку предшественники являются *частью* идентичности.

Из-за практических недостатков записи ходов в виде цепочки мы решили моделировать систему как имеющую отдельные ходы, каждый из которых имеет свой индекс. Таким образом, мы отказываемся от рис. 3.22 и возвращаемся к рис. 3.21. Это оставляет задачу проверки индексов, чтобы убедиться в отсутствии пробелов и дубликатов. Мы можем просто добавить это к проверке, которая уже должна происходить для защиты от незаконных действий, таких как переход к проверке. Модель может развиться даже до того, чтобы сделать недействительное положение непредставимым.

Поскольку ходов в партии становится все больше и больше, представление их в виде отдельных фактов на диаграмме становится утомительным. Поэтому вместо этого мы группируем их вместе. У группы есть общий предшественник – партия. Внутри рамки мы можем просто нарисовать набор ходов в виде таблицы, как показано на рис. 3.23.

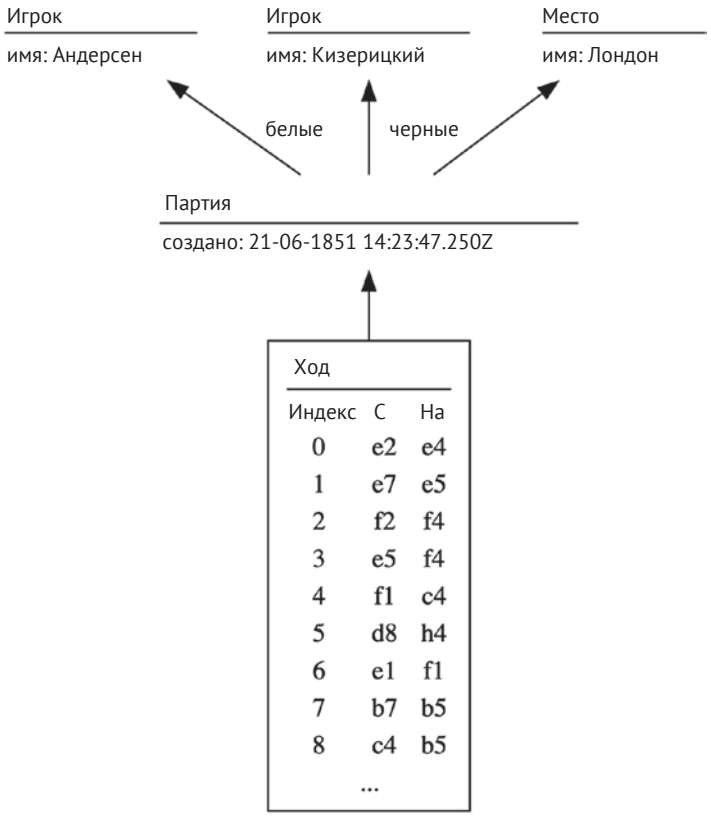


**Рис. 3.23.** Ходы группируются вместе под общим предшественником

Группировка не меняет того факта, что каждый ход является отдельной записью. Порядок ходов в рамках группировки не подразумевает никакого отношения между фактами. Это просто удобство, чтобы ограничить размер иллюстрации.

## Блестящая победа

В партии, которую мы моделируем, Андерсен блестяще пожертвовал, чтобы добиться победы белыми. Мы представим эту победу на рис. 3.24 с помощью факта, который включает партию, победителя и набор ходов в качестве предшественников.

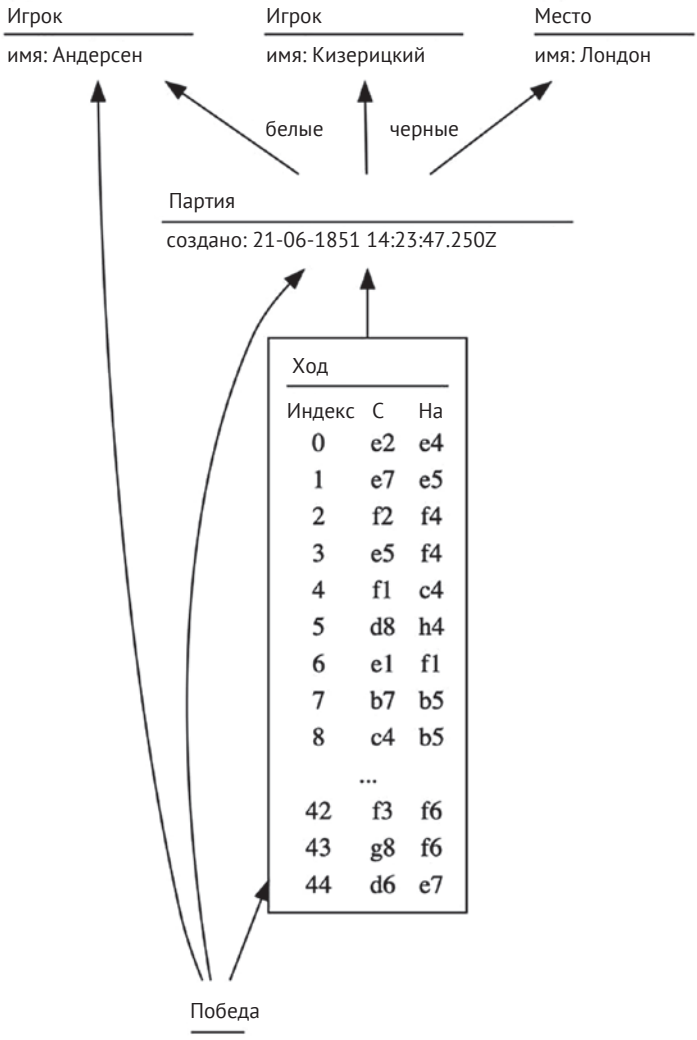


**Рис. 3.24.** Победа охватывает несколько предшественников, включая все ходы партии

Факт победы не имеет полей. Ему не нужна дополнительная информация. Он говорит все, что ему нужно сказать, с помощью предшественников, которых он собирает вместе. Мы изображаем предшественников в виде набора ходов с помощью стрелки, указывающей на всю группу. Каждый ход в группе является предшественником хода победы.

Как показывает этот пример, графы экземпляров фактов значительно отличаются от графов типов фактов. Они представляют отдельные экземпляры фактов, с их типом и полями. Они показывают отношения предшественник/преемник между фактами, а не роли между типами. Если граф типов фактов допускает циклы между типами, то граф экземпляров не допускает циклов между экземплярами. Цикл в графе типов фактов разворачивается в цепь в графе экземпляров фактов.

Графы типов фактов и графы экземпляров фактов используются для разных целей. Граф типов фактов делает общие утверждения обо всех возможных моделях данных типов фактов. Мы используем его для рассуждений о правилах области. Граф экземпляров фактов, с другой стороны, иллюстрирует конкретный пример набора данных. Мы используем его, чтобы попробовать различные сценарии и отладить модель перед ее реализацией.

Поскольку графы типов фактов лучше описывают общие правила области, мы будем полагаться на них больше, чем на графы экземпляров фактов. Графы экземпляров фактов часто слишком специфичны, чтобы делать общие заявления. В оставшейся части этой книги они будут использоваться только для иллюстрации конкретных закономерностей.

## Язык фактологического моделирования

Хотя визуальный язык исторического моделирования полезен для понимания отношений внутри модели, он недостаточен для строгих рассуждений. Он не может определить модель до такой степени, чтобы можно было доказать утверждения или сгенерировать код. Чтобы удовлетворить эту потребность, мы используем язык фактологического моделирования (Factual Modeling Language), или Factual.

С ним мы можем писать точные спецификации. Он описывает все данные, которые являются частью модели, способы, которыми эти элементы данных связаны между собой, и правила, по которым их можно запрашивать. Он даже описывает правила, по которым данные защищаются и проверяются. Этот точный язык спецификаций позволяет нам рассуждать о требованиях системы и заблаговременно определить, что мы сможем реализовать.

### Объявление типов фактов

Типы фактов объявляются в Factual с помощью ключевого слова `type`. Тело типа, заключенное в скобки, включает список полей и предшественников. Поля объявляются в стиле, напоминающем Pascal и родственные языки программирования, – за именем поля следует двоеточие и тип. Поля имеют собственные типы данных, такие как `string`, `int` и `bool`.

```
fact Catalog {
  referenceNumber: string
}
```

Предшественники объявляются аналогичным образом. Основное различие заключается в том, что предшественник ссылается на другой тип факта. Предшественники могут появляться до, после или перемежаться с полями.

```
fact Product {
  catalog: Catalog
  sku: string
}
```

Индикаторы кардинальности модифицируют объявления предшественников. Предшественники единственного числа не имеют модификатора (как было показано ранее). Необязательные предшественники объявляются с модификатором в виде вопросительного знака, а множественные предшественники объявляются со звездочкой.

```
fact InvoiceLine {
  orderLine: OrderLine?
  total: decimal
  description: string
}

fact Invoice {
  lines: InvoiceLine*
  subtotal: decimal
  tax: decimal
}
```

Предшественник может ссылаться на тип, который еще не был объявлен. Он может даже ссылаться на тип, в котором он объявлен. Такие самореферентные предшественники, как мы уже обсуждали ранее, часто используются для ссылки на предыдущие версии изменяемых свойств. Помните, что хотя это вводит цикл в графе типов, это не допускает циклов в графе экземпляров:

```
fact Price {
  product: Product
  value: decimal
  prior: Price*
}
```

Синтаксис объявления фактов разработан таким образом, чтобы быть знакомым разработчикам. Он также достаточно прост, чтобы быть механизмом общения между разработчиками и не разработчиками. Он не содержит ни поведения, ни модификаторов доступа, ничего, что можно было бы спутать с кодом. И тем не менее он достаточно точен и выразителен, чтобы описать основы модели.

Рассмотрим каждый факт как решение, которое принимает человек или другой субъект. Факт отражает детали этого решения. Он также показывает, какие решения были приняты до него, в виде предшественников. Проходя через процесс создания фактов, аналитики рассказывают о том, как система развивается для решения бизнес-задач.

## Запрос модели

Как бы мощно ни было объявлять типы фактов в системе, также полезно отвечать на вопросы, основанные на этих фактах. Язык фактологического моделирования включает синтаксис запросов к модели для определения ее текущего состояния. Запросы объявляются с помощью ключевого слова `query` и имеют описательное имя. Все запросы начинаются с точки зрения данного факта, который задается в качестве параметра запроса.

```
query productsInCatalog(c: Catalog) {
  match p: Product where p.catalog = c
}
```

Тело запроса включает ключевое слово `match`. Это вводит условие, которое определяет другие факты, связанные с начальной точкой. Укажите псевдоним и тип факта, который нужно определить, а затем предложение `where`. После `where` приравняйте предшественников каждой стороны.

Предыдущий запрос находит все продукты в пределах заданного каталога. При наложении на диаграмму типа факта запрос можно представить как прохождение отношения предшественника в обратном направлении. На рис. 3.25 это показано пунктирной линией.

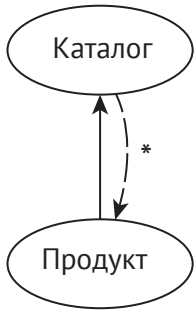


**Рис. 3.25.** Запрос находит всех преемников предшественника

Поскольку он следует отношениям преемника, вышеупомянутый запрос может вернуть несколько результатов. Я пометил пунктирную линию звездочкой, чтобы проиллюстрировать этот момент. Большинство запросов будут включать пути преемников и, следовательно, возвращать несколько результатов.

## Переход по уровням

Предложение `where` не ограничивается следованием по одной ссылке предшественника. С помощью цепочки дополнительных ссылок предшественников запрос может продвинуться дальше по графу преемников. Например, строка в заказе фиксирует конкретную цену продукта. В модели, которую мы определили ранее, продукт является предшественником цены. Поэтому строка заказа может достичь продукта только косвенно через факт цены. Чтобы найти все строки заказа для конкретного продукта, мы обходим эти два отношения в обратном направлении.

```
query orderLinesForProduct(p: Product) {
  match ol: OrderLine where ol.price.product = p
}
```

Поскольку связь между строкой заказа и продуктом является косвенной, запрос обращается к промежуточному предшественнику. Начиная с продукта, мы находим все строки заказа, где цена продукта является отправной точкой. Давайте наложим запрос на диаграмму типов фактов, как показано на рис. 3.26, чтобы увидеть, как происходит переход по уровням.

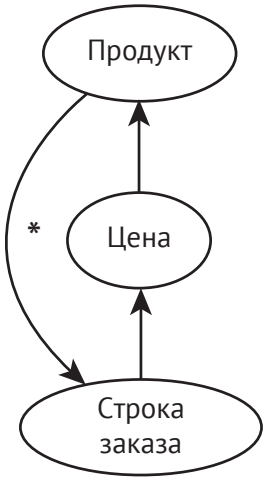


**Рис. 3.26.** Запрос находит преемников на два уровня ниже

## Объединение совпадений

Помимо углубления в граф, мы также можем оттолкнуться от него. Добавляя дополнительные совпадения в запрос, можно пройти вверх по другой ветви графа. Используйте ключевое слово `then` вместо `match`, чтобы указать на присоединение. Мы можем использовать это, например, чтобы найти все корзины, содержащие заданный продукт.

```
query cartsContainingProduct(p: Product) {
  match ol: OrderLine where ol.price.product = p
  then c: Cart where c = ol.cart
}
```

Первое совпадение находит все строки заказа, которые косвенно ссылаются на данный продукт. Второе совпадение находит корзину, содержащую эту строку заказа. Для этого нужно просто следовать по ссылке на предшественника. Полный запрос, переходящий вниз к строкам заказа, а затем обратно вверх, к корзинам, показан на рис. 3.27.

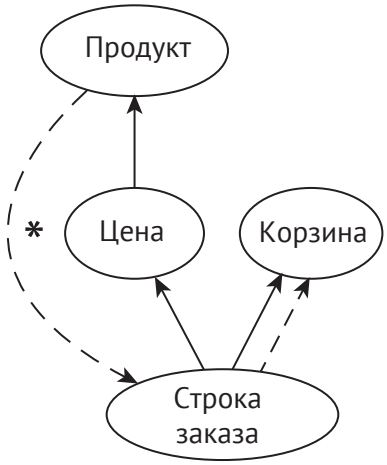


**Рис. 3.27.** Запрос выбирает предшественников промежуточных преемников

Поиск по преемникам может вернуть много результатов. Я проиллюстрировал это с помощью индикатора количества (*). Но соответствие, идущее вверх к предшественнику, вернет только одну корзину для строки заказа. Эта стрелка не имеет индикатора количества. В целом запрос вернет несколько результатов, поскольку он включает в себя совпадение с преемником.

Синтаксис запроса, который мы только что описали, достаточен для обхода вниз любого количества преемников, вверх – любого количества предшественников и отталкивается от фактов, чтобы следовать в разных направлениях. Это охватывает весь граф связанных фактов. Если один факт каким-то образом связан с другим, то можно написать запрос, чтобы найти набор, включающий один факт, начиная с другого.

## Экзистенциальные квантификаторы

Часто мы сталкиваемся с необходимостью ограничить количество фактов, соответствующих запросу. Вместо того чтобы перечислять все связанные факты, мы хотим сосредоточиться на подмножестве, которое находится в определенном состоянии. Состояние факта не является неотъемлемой частью самого факта, факты сами по себе неизменны. Скорее, состояние факта определяется

наличием или отсутствием преемников. Поэтому мы ограничиваем запросы с помощью экзистенциальных квантификаторов.

Экзистенциальный квантификатор имеет форму выражения `such that not exists`. Выражение включает в себя алиас, тип и условие `where` так же, как и условие `match`. Однако вместо того чтобы *вернуть* все совпадающие факты, это предложение *исключает* совпадение, если оно существует.

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
  such that not exists o: Order where o.orderLines = ol
}
```

Предыдущий запрос находит все строки в корзине, которые еще не являются частью заказа. Он показывает только незаказанные строки, что может быть полезно для обновления пользовательского интерфейса или создания нового заказа. Когда создается заказ, ссылающийся на строку заказа, она удаляется из результатов запроса. Запрос, дополненный предложением `not exists` (-∃), показан на рис. 3.28.


**Рис. 3.28.** Запрос определяет преемников, для которых не существует преемника второго уровня

Экзистенциальный квантификатор может быть полезен и в других запросах. Вы можете объявить его независимо от запроса с помощью ключевого слова `predicate`. Например, было бы эквивалентно назвать предыдущий предикат `orderLineIsOrdered`.

```
predicate orderLineIsOrdered(ol: OrderLine) {
  exists o: Order where o.orderLines = ol
}
```

Предикат относится только к экзистенциальному квантификатору, показанному на рис. 3.29.

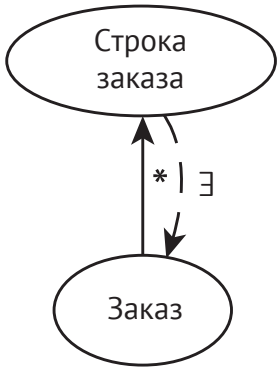


**Рис. 3.29.** Предикат, проверяющий наличие преемника

Предикат является ложным для строки заказа до тех пор, пока не создан заказ-преемник. Как только заказ будет создан, предикат станет истинным. Теперь мы можем ссылаться на этот предикат по имени в запросе.

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
    such that not orderLineIsOrdered(ol)

}
```

Этот запрос имеет тот же эффект, что и исходный. Предикат был просто извлечен и назван так, чтобы его можно было использовать повторно.

## Текущее значение

При таком использовании экзистенциальные квантификаторы помогают нам определить состояние факта в рамках рабочего процесса. Используемые немного иначе, они также помогают определить текущее значение изменяемого свойства. Текущая цена продукта – это цена, которая не была заменена следующей ценой.

```
query currentPriceOfProduct(pr: Product) {
  match p: Price where p.product = pr
    such that not exists next: Price where next.prior = p

}
```

Действительно, я не могу гарантировать, что этот запрос вернет только одну цену. Хотя мне хотелось бы верить, что у продукта есть только одна текущая цена, модель не позволяет мне считать ее единственной. Я должен написать этот запрос как множество текущих цен. Это, как правило, справедливо для всех запросов к преемнику; число их результатов всегда равно нулю или боль-

ше. Как показано на рис. 3.30, запрос включает в себя соединение с преемником (отмечено звездочкой).

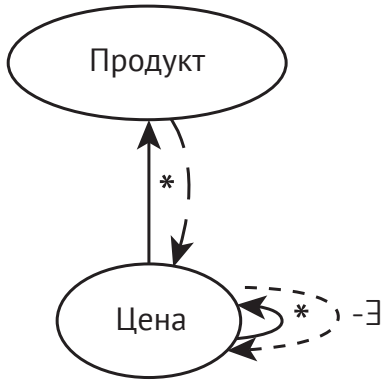


**Рис. 3.30.** Текущая цена продукта – это цена, для которой не существует преемника

У изменяемого свойства есть ноль или более текущих значений. Если возвращается только одно значение, то его изменений не было. Но когда возвращается более одного значения, то это значения-кандидаты. Рассмотрим изменяемые свойства более подробно.

## Правила авторизации

Важным аспектом модели является то, кто уполномочен выполнять те или иные действия. Поскольку факт соответствует решению, которое принял человек, факты являются идеальным средством для определения действий, которые может выполнять пользователь. Поэтому мы моделируем правила авторизации как запросы, начинающиеся с авторизованного факта.

Предположим, например, что пользователь создал корзину. Мы можем зафиксировать личность пользователя в качестве предварительного условия.

```
fact Cart {
  createdBy: User
  createdAt: timestamp
}
```

Добавление строки заказа в корзину может быть выражено в виде правила авторизации. Только создателю корзины разрешено добавлять строку заказа. Это выражается с помощью ключевого слова `authorize`.

```
authorize ol: OrderLine {
  match u: User where ol.cart.createdBy = u
}
```

Тело правила авторизации точно такое же, как тело запроса, который возвращает коллекцию пользователей. Если пользователь, который инициировал действие, находится в результатах запроса, то этот пользователь авторизован для выполнения действия. Мы обсудим авторизацию более подробно в главе 7.

Мы будем использовать как визуальный, так и текстовый язык исторического моделирования на протяжении всей оставшейся части этой книги. Визуальный язык будет основным инструментом для понимания модели. Больше всего мы будем опираться на графы типов. С другой стороны, текстовый язык будет использоваться для описания модели с полезной степенью детализации. Эта форма поможет нам рассуждать и доказывать утверждения о поведении системы.

Применяя историческое моделирование в своей работе, вы обнаружите, что начинаете с визуального языка. Он весьма полезен для первоначального планирования модели и демонстрации отношений между фактами. Однако вы быстро перейдете к текстовому языку для реализации модели. Инструменты, которые вы будете использовать, преобразуют текстовый язык в код и графы. Код станет основой для реализации вашей системы, а графы будут визуально документировать модель. Язык фактологического моделирования будет вашим инструментом для построения рабочих моделей и обмена ими с другими.

## Шахматное приложение

Теперь, когда мы проанализировали модель шахматной партии и построили ее пример, давайте посмотрим, как это будет выглядеть в виде спецификации Factual. Цель этой спецификации – предоставить достаточную детализацию для написания требований к приложению и создания реализации. Мы можем начать с записи типов фактов, которые уже определили визуально.

```
fact Player {
  name: string
}

fact Place {
  name: string
}

fact Game {
  white: Player
  black: Player
  place: Place
  createdAt: timestamp
}

fact Move {
  game: Game
  index: int
  from: int
  to: int
}
```

```
fact Win {
  game: Game
  player: Player
  moves: Move*
}

fact Draw {
  game: Game
  players: Player*
  moves: Move*
}
```

Разработчики среди вас, вероятно, уже знают, как генерировать типы данных, API и таблицы баз данных. Подождите, мы еще дойдем до этого. Однако самое главное – это то, что бизнес-аналитики среди вас также могут понять эту модель. Все слова, за исключением примитивных типов и ключевого слова `fact`, написаны на языке доменной модели. Возможно, у вас останется вопрос типа «Как квадрат представляется в виде целого числа?», но другие вопросы, специфичные для реализации, не возникают в этой модели, ориентированной на предметную область.

## Примеры использования

Учитывая только описания фактов, команда уже может приступить к обсуждению вариантов использования модели. Какие действия может предпринять пользователь системы? Каждый факт – это сценарий использования.

Факт «Игрок» (Player), например, генерируется, когда в системе регистрируется новый игрок. Единственное поле в Player – это имя, что означает, что игрок однозначно идентифицируется только по имени. Это также означает, что игрок не может изменить свое имя. Если данные условия неприемлемы, сейчас самое время изменить модель.

Факт партии (Game) рассказывает о другом случае использования – начале партии. Для того чтобы начать партию, нужны два игрока в одном месте. Это вызывает дискуссию о том, один из игроков начинает игру или, возможно, организатор турнира в определенном месте. Каковы правила для определения того, кто играет белыми? Все эти решения должны быть приняты до записи факта партии.

Модель также показывает, что нам нужен пример использования, описывающий, как игрок делает ход (Move). Он должен включать правила проверки, например играющий белыми делает четные индексированные ходы, а играющий черными – нечетные индексированные ходы. Другие правила проверки описывают, что делать с дублирующимися индексами или пробелами. Возможно, нам также нужно добавить некоторые правила авторизации, например ход может быть сделан только игроком данной партии.

```
authorize m: Move {
  m.game.white
}

authorize m: Move {
  m.game.black
}
```

Пройдя через эти примеры использования с моделью в руках, мы можем оценить последствия своего выбора. При необходимости мы можем внести изменения в модель, чтобы получить желаемые возможности. Все это можно сделать до того, как мы реализуем первую пользовательскую историю. Более подробно мы обсудим это в главе 5.

## Интерфейс пользователя

После того как мы убедились, что зафиксировали нужные факты, мы можем приступить к их отображению в пользовательский интерфейс. Именно здесь язык моделирования Factual прекрасно сочетается с информационной архитектурой, картами сайта, макетами и схемами. Factual выражает два аспекта этого дизайна – какие действия может предпринять пользователь и что он может увидеть.

### Действия

Во-первых, рассмотрим действия, которые пользователь может предпринять в отношении исторических фактов. Вы можете определить, что вам необходимо обеспечить действия по их созданию, обновлению и удалению, и поэтому спроектировать модель CRUD[1]. Этому способствуют структурные шаблоны, которые мы обсудим в главе 8. Однако более вероятно, что вам нужен пользовательский интерфейс, ориентированный на задачу. Пользователь собирает достаточно информации для решения задачи. Эта задача затем фиксируется как факт.

В случае с шахматным приложением мы можем решить, что факт `Place` лучше всего поддерживать с помощью интерфейса CRUD. Администратору турнира нужна возможность создавать, обновлять и удалять места. Для поддержки такого интерфейса нам потребуется расширить модель. Например, поле `name` должно стать изменяемым свойством. И должен быть введен новый факт для отображения удаления места.

Другие части пользовательского интерфейса должны быть немного более ориентированы на конкретные задачи. Начало партии – это целая задача, в которую вовлечены и игроки, и место. Выполнение хода – это основная задача игрового интерфейса. Для каждого действия, которое может выполнить пользователь, запишите факт, который будет создан в ответ.

[1] CRUD – широко распространенный термин, означающий 4 стандартные операции над любой сущностью (ресурсом): создание (create), чтение (read), обновление (update) и удаление (delete). – *Прим. перев.*

## Представления

Второй аспект дизайна пользовательского интерфейса, который может быть выражен в Factual, – это то, что может видеть пользователь. Подумайте о том, как пользователь задает вопросы модели и как пользовательский интерфейс будет отвечать на эти вопросы. Затем свяжите области модели с запросами на языке моделирования Factual.

Подумайте о том, что увидит игрок, когда войдет в систему. В этот момент у нас есть факт `Player`. Мы можем отобразить список партий, которые в данный момент играются.

```
query gamesInProgress(p: Player) {
  match g: Game where g.white = p or g.black = p
    such that not exists w: Win where w.game = g
    and not exists d: Draw where d.game = g
}
```

В углу мы можем отобразить счет игрока. Победа оценивается в одно очко, а ничья стоит пол-очка. Это вычисляется с помощью следующих запросов:

```
query wins(p: Player) {
  match w: Win where w.player = p
}

query draws(p: Player) {
  match d: Draw where d.player = p
}
```

Укажите в модели, что количество очков для отображения равно 1*wins(p).count + 0.5*draws(p).count. Это обеспечивает достаточную точность, чтобы полностью определить желаемое поведение. Это также подразумевает, что когда фиксируется победа или ничья, количество очков должно быть обновлено.

Фактологический язык моделирования обеспечивает достаточную детализацию для выражения этих требований. Рассуждая на этом языке, бизнес-аналитики, разработчики и проектировщики могут определить последствия своих решений. Когда они изменяют решение для получения лучших результатов, они могут увидеть, как это изменение повлияет на другие части системы.

Каждая из описанных здесь форм выражения имеет свое место. Обычно мы начинаем с чистого листа, внося быстрые изменения в модель с помощью графов типов фактов. Это фиксирует важные объекты и действия пользователя и показывает, как они связаны между собой. Затем мы записываем несколько примеров, используя графы экземпляров фактов. Это помогает проверить модель, по мере того как мы исследуем конкретные сценарии. И наконец, мы потратим время на то, чтобы выразить на языке моделирования фактов конкретную структуру, запросы и правила авторизации системы. Это гарантирует, что мы полностью проанализировали, задокументировали и можем реализовать систему точно.

Все эти элементы поддерживают общение в команде. Графические или текстовые формы, в которых выражается модель, предназначены для создания

и потребления всеми членами команды. Они предназначены для того, чтобы сосредоточиться исключительно на проблемной области, а не на каких-либо конкретных деталях реализации. И все же, поскольку эти элементы подчиняются законам неизменяемой архитектуры, они приведут к продуманной реализации.

Давайте теперь сосредоточимся на этих деталях реализации. Типы систем, которые мы проектируем с использованием неизменяемой архитектуры, будут, как правило, распределенными системами. Давайте изучим природу распределенных систем и посмотрим, почему неизменяемые архитектуры, подобные той, которую мы описали здесь, так хорошо работают в этом контексте.

# ЧАСТЬ II

# Применение

# Глава 4

# Независимость от местоположения

В недалеком прошлом большинство программ выполнялись на одном компьютере. После распространения JavaScript в веб-браузере, приложений на мобильных телефонах и микросервисов в облаке большинство программ, которые мы пишем сегодня, работают на многих компьютерах. В то время как распределенные системы раньше были специализацией, сегодня они используются по умолчанию. Нам необходимо обновить другие параметры по умолчанию, чтобы удовлетворить эту потребность.

Одно из значений по умолчанию, которое нам необходимо обновить, – это предположение о том, что данные имеют местоположение. Некоторые системы пытаются обращаться с удаленными объектами так, как будто они локальные. DCOM использует идентификаторы объектов, чтобы прокси-сервер выглядел как локальный экземпляр удаленного объекта. Удаленные вызовы процедур (RPC) пытаются скрыть реальность сетевого взаимодействия за интерфейсом, который выглядит как обычная функция. Проблемы с этими системами были хорошо освещены в статьях, поэтому я не буду повторять их здесь.

Предположения о локальности, которые я хочу рассмотреть, немного более тонкие. Даже когда мы заменяем RPC на сообщения, а идентификаторы объектов на URL, легко предположить, что у данных есть местоположение. Мы делаем это предположение всякий раз, когда определяем «источник истины» или «систему записи». Мы полагаемся на местоположение всякий раз, когда один узел генерирует уникальные идентификаторы. Наш стандартный способ программирования того, что происходит на компьютере, просачивается в поведение, которое мы программируем для системы в целом.

Очень многие из моделей поведения, которые мы привыкли ожидать от наших систем, зависят от местоположения. Мы ожидаем, что предметы будут последовательно упорядочены. Мы ожидаем, что система будет отклонять повторяющиеся имена. Мы ожидаем, что когда пользователь обновляет свойство объекта, оно будет иметь то значение, которое он только что присвоил. На самом деле даже ожидание того, что свойства имеют единственное значение, является предположением, зависящим от местоположения.

Система, которая зависит от местоположения, будет вести себя неправильно, когда оно становится недоступным. Если вместо этого мы будем стремить-

ся к независимости от местоположения, то создадим системы, которые будут более отзывчивыми, устойчивыми и надежными. Они смогут действовать автономно без общения с удаленными узлами, смогут переносить сбои в сети, не внося дефектов. И решения, которые пользователь принимает в изоляции, будут выполняться, когда другие узлы и пользователи узнают об этом решении.

## Моделирование с неизменяемостью

В своей основе предположение о местоположении связано с возможностью изменения. Переменная – это место, которое хранит значение. Программы используют переменную, чтобы обратиться к этому месту и прочитать ее значение. После обновления переменной мы ожидаем, что программа прочитает новое значение в следующий раз, когда обратится к ней. Масштабируйте это до уровня распределенной системы, и вы получите зависимость от местоположения. Система зависит от данных, находящихся в определенном месте, именно потому, что эти данные могут меняться.

Если вместо этого мы ищем модель вычислений, основанную на неизменяемости, то зависимость от местоположения исчезает. Если объект не может измениться, то каждая его копия будет одинаково хороша. Нет необходимости знать, где хранится объект, где он был создан или какая подсистема является источником истины.

Конечно, нам нужно моделировать области, которые изменяются во времени. Таким образом, понятия времени и изменения должны быть пересмотрены в свете неизменяемости.

## Синхронизация

Нередко об управлении данными в распределенных системах говорят как о проблеме синхронизации. Но даже этот термин происходит из места размещения данных. Синхронизация – это задача изменения данных в двух или более местах в более или менее одинаковое время. Когда данные в двух местах различаются, эти места рассинхронизованы. Система, зависящая от местоположения, будет стремиться синхронизировать их.

Когда данные больше не зависят от местоположения, одновременные изменения становятся возможными. Временное рассогласование между двумя узлами – это не проблема синхронизации, которую нужно решить, а возможность для них со временем совпасть между собой. Независимая от местоположения система использует другое определение времени, чтобы можно было описать параллелизм. Она смягчает свои предположения, так что изменения больше не являются линейными. Это помогает системе гарантировать, что одновременные изменения не вызывают конфликтов.

Система, не зависящая от местоположения, связана не с синхронизацией, а с причинностью. Она стремится понять, какие события вызвали какие другие события. Там, где синхронизация описывает согласование структур данных, хранящихся в разных местах, причинность описывает историю самих данных, независимо от того, где они хранятся. Причинность является более слабым ограничением, чем синхронизация, но ее гораздо легче осуществить.

# Изучение соглашений

В этой главе мы рассмотрим некоторые важные соглашения, которые может соблюдать распределенная система. Среди них будут такие простые и очевидные, как «я ожидаю, что прочитаю то, что я только что написал». Мы также увидим некоторые очень неуловимые и мощные соглашения, такие как «все узлы в конечном итоге приходят к одному и тому же состоянию». Мы посмотрим, что вообще нужно сделать, чтобы выполнить одно из этих соглашений. Попутно мы будем искать компромиссы, отказываясь от одних гарантий, чтобы получить другие.

Мы обнаружим, что соглашение, которое, как мы думаем, нам нужно – которое эквивалентно предположению, что у данных есть местоположение, – недостижимо во многих ситуациях. Вместо этого соглашение, на которое мы обменяем эту гарантию, будет таким, которое позволяет получить доступ к данным независимо от местоположения. Если мы распознаем, когда мы сделали предположение, что данные хранятся в определенном месте, мы можем выбрать другое соглашение.

# Идентичность

Первой задачей в поисках независимости от местоположения является отделение идентичности объекта от места его хранения. Когда местоположение является частью идентичности, объекты имеют определенное сродство с компьютерами в этом месте. Для достижения наилучших результатов пользователи должны иметь возможность легко идентифицировать объекты из любого места, без необходимости общения.

## Автоинкрементные идентификаторы

При работе с реляционной базой данных можно встретить автоинкрементные идентификаторы. Большинство систем управления базами данных включают механизм для их генерации при вставке. Наиболее распространенным способом идентификации объекта является использование номера, который был сгенерирован при вставке объекта в таблицу.

Автоинкрементные идентификаторы – отличный способ создания уникальных первичных ключей. Они монотонно возрастают, что делает их идеальными для кластерных индексов. Вы никогда не разделите страницу при вставке новой записи с возрастающим первичным ключом. Это идеальные внешние ключи, гораздо более компактные, чем любой другой столбец, который может иметь ссылочная таблица. И они не изменяются. Большинство систем управления базами данных принимают меры предосторожности, чтобы предотвратить обновление автоинкрементных столбцов. Это помогает сохранить их уникальность и полезность в качестве ссылок.

Кажется, что автоинкрементный идентификатор является идеальным идентификатором для объекта в системе, основанной на реляционной базе данных. Но хотя эти идентификаторы идеально подходят для представления идентичности *внутри* базы данных, они плохо подходят для расширения идентичности *за ее пределы*. Удобство такого подхода сделало их выбором по умолчанию для идентификации, но в дальнейшем это вызвало множество проблем.

Их суть заключается в том, что автоинкрементный идентификатор генерируется в определенном месте. Он имеет значение только в пределах одной базы данных. Хотя верно, что эта единственная база данных может быть кластеризована и распределена по нескольким компьютерам, она все равно логически является одним местоположением. Она доступна по единственной строке соединения и может легко стать недоступной для удаленных клиентов.

### Зависимость от среды

Если вы когда-либо переносили программное обеспечение из среды разработки в среду тестирования, реализации, а затем в эксплуатацию, вы хорошо знаете, что идентификаторы, созданные в каждой среде, не передаются в другие. Объект, получивший ID 1337 в тестовой среде, не будет таким же, как объект 1337 в эксплуатации. Это может вызвать легкое раздражение, когда вы создаете резервную копию базы данных тестирования для восстановления в процессе разработки, чтобы воспроизвести ошибку. После восстановления все идентификаторы относятся к одним и тем же объектам в каждой среде. Но когда вы начинаете работать с одной или другой системой, идентификаторы начинают расходиться. Это означает, что вы не можете легко импортировать больше данных из тестирования, не учитывая базу данных разработки.

Это становится проблемой при перемещении данных между реализацией и эксплуатацией. Распространенной практикой является резервное копирование данных эксплуатации и их восстановление в среде реализации. Затем вы можете обновить рабочую среду до последней версии программного обеспечения, одновременно применяя все необходимые обновления базы данных. После быстрого тестирования вы убеждаетесь, что развертывание прошло успешно. В этот момент было бы желательно просто переместить среду реализации в эксплуатацию, но это не сработает, если во время этого процесса не была отключена эксплуатация. Если она все еще была запущена, то, скорее всего, новые данные были вставлены в производственную базу данных, получив новые автоинкрементные идентификаторы. Эти идентификаторы не имеют смысла в базе данных реализации.

Автоинкрементные идентификаторы из раздражителя становятся препятствием, когда мы пытаемся реализовать решение для аварийного восстановления в режиме теплого резерва. Цель состоит в том, чтобы иметь копию производственных данных в географически изолированном центре данных для защиты от локального сбоя. До того, как произойдет отключение, записи, хранящиеся в производственной базе данных, отправляются в удаленную базу данных с минимально возможной задержкой. Задержка должна быть малой, чтобы обеспечить минимальную потерю данных в случае сбоя. Когда происходит сбой, приложение должно «переключиться» на удаленную реплику. Перед преодолением отказа за генерацию идентификаторов отвечает производственная база данных. После преодоления отказа ответственной становится удаленная база данных.

Так же, как задержка при передаче данных должна быть малой, время, необходимое для преодоления отказа, должно быть малым. К сожалению, задержка не может быть нулевой, и переключение никогда не может быть мгновенным. Трудно добиться правильного выбора времени импорта всех

производственных данных до включения генерации идентификаторов. Сокращение задержки, особенно между географически разнесенными точками, становится тем дороже, чем ближе мы подходим к нулю. Потеря данных во время аварийного восстановления может быть еще более дорогостоящей. И чем дольше мы ждем данных, тем дольше приходится откладывать генерацию новых идентификаторов.

Я участвовал во многих длительных и дорогостоящих проектах по организации аварийного восстановления. Некоторые из них даже были успешными. После нескольких фальстартов нам удавалось добиться надежного восстановления системы. Но «возврат назад» – это гораздо более сложная задача. После решения первоначальной производственной проблемы нам приходилось запускать весь процесс в обратном направлении. Я никогда не видел, чтобы это делалось без отключения системы на длительный период времени. Это было бы намного проще, если бы мы не возлагали на базу данных дополнительное бремя создания идентификаторов для конкретного местоположения.

## Вставка отношения «родитель–ребенок»

Неудобство использования автоинкрементного ID в качестве идентификатора становится очевидным при работе с отношениями «родитель–ребенок». Родительская запись имеет первичный ключ. Дочерние записи имеют внешний ключ. База данных обеспечивает ссылочную целостность внешних ключей, поэтому родительская запись должна быть вставлена раньше дочерних. Вставка дочерних записей не может начаться, пока не завершится вставка родительской записи и не будет получен автоинкрементный идентификатор.

Мы нечасто думаем о том, что база данных и приложение – это два разных места, но на самом деле так оно и есть. Приложение создает инструкции вставки (INSERT) и передает их в базу данных для выполнения. При нормальных обстоятельствах приложение может создать несколько операторов INSERT и попросить базу данных выполнить их пакетно. Но в случае отношений «родитель–ребенок» приложение должно подождать, пока завершится вставка родителя, прежде чем оно сможет узнать первичный ключ. Только после этого оно может генерировать партию дочерних вставок.

Объектно-реляционные картографы (object relational mappers – ORM) выполняют, помимо прочего, задачу вставки отношений «родитель–ребенок». Со стороны это выглядит так, как будто мы можем построить граф объектов, а затем выполнить единственную команду для сохранения изменений. Но внутри ORM эта единственная операция распределяется на несколько пакетов команд INSERT, отправляемых в базу данных в строго определенном порядке.

ORM скрывают это поведение от приложений настолько хорошо, насколько это возможно, но оно все же просачивается через абстрагирование. Когда объект представляет первичный ключ в качестве свойства, чтобы использовать его как внешний идентификатор, этот первичный ключ изначально равен нулю или отрицателен. После команды сохранения объектов в базе он становится положительным. Предполагается, что первичный ключ объекта не должен меняться, но необходимость перехода в другое место для генерации автоинкрементного идентификатора заставляет ORM нарушать эту инвариантность.

Когда приложение близко к своей базе данных, мы можем попытаться скрыть правду об автоинкрементных идентификаторах в ORM. Но по мере удаления узла от центральной базы данных зависимость от местоположения становится сложнее скрыть.

### Удаленное создание

Рассмотрим мобильное приложение. Оно имеет локальную базу данных для хранения копии пользовательских данных, для быстрого доступа к ним, даже если устройство находится в медленной сети. Предположим далее для простоты, что эти локальные данные имеют такую же схему, как и центральная база данных.

Когда устройство получает данные из центрального приложения, оно сохраняет объекты с предоставленными идентификаторами. После этого оно может быстро представить эти данные, выполнив локальные запросы к своей собственной копии. Пользователь может даже вносить изменения. Эти изменения сначала применяются к локальной копии, а затем сохраняются в очереди для отправки в центральное приложение.

Все работает хорошо для запросов и обновлений. Но проблема возникает, когда мы пытаемся вставить новые объекты. Локальная база данных не может использовать автоинкрементный ID для создания новых записей. Если бы она это сделала, то часто генерировала бы ID, который центральная база данных уже использовала для другого объекта. Таким образом, если бы автоинкрементный ID был использован в качестве идентификатора объекта, приложению пришлось бы обратиться к центральной базе данных, чтобы получить корректный ID.

По этой причине простое решение часто не используется в мобильных приложениях. Вместо этого они выбирают локальную базу данных, которая не так сильно полагается на внешние ключи. Это, по крайней мере, позволяет мобильному клиенту создавать целые структуры объектов до того, как он узнает их идентичности. Такой подход откладывает проблему идентификации по местоположению достаточно далеко для большинства приложений, но это не полное решение. Полное решение позволило бы полностью удалить из идентичности компонент, зависящий от местоположения, такой как автоинкрементный идентификатор.

## URL-адреса

Веб-приложения, которые следуют архитектурному стилю REST, обычно используют гипертекст как механизм состояния приложения (Hypertext as the Engine of Application State – HATEOAS). Каждая операция, выполняемая приложением, является запросом к ресурсу. С каждым запросом приложение переходит в другое состояние. Когда гипертекст используется в качестве двигателя этого состояния, идентичности доступных ресурсов возвращаются в виде ссылок в каждом ответе.

Идентичность в архитектурном стиле REST определяется унифицированным идентификатором ресурса (URI). Это иерархическая идентичность, так что генератор может гарантировать, что новые URI будут уникальными. Общепринятой практикой является использование доменного имени гене-

ратора в качестве первого уровня этой иерархии. Доменное имя идентифицирует небольшую коллекцию узлов, которые часто являются близко расположенными.

Для того чтобы приложение выбрало и выдало следующую команду (и таким образом перешло в следующее состояние), ему необходимо каким-то образом отправить команду на нужный узел. По этой причине URI, используемые в HATEOAS, часто являются не просто идентификаторами, а унифицированными указателями ресурсов (Uniform Resource Locators – URL). URL имеет ту же иерархическую структуру, что и URI, но у него есть дополнительное ограничение. URL должен быть адресуемым. Он должен нести достаточно информации, для того чтобы приложение могло послать команду на хост, который ее выполнит.

URL несут доменное имя не только как пространство имен идентификации, но и для того, чтобы клиент мог преобразовать доменное имя в IP-адрес. Этот IP-адрес должен быть способен направить последующую команду на узел, который ее выполнит. Таким образом, доменное имя тесно связано с местоположением ресурса.

Когда URL-адреса используются в качестве идентификатора ресурсов, может быть очень сложно переместить ресурс из одного места в другое. Либо новое место должно быть адресуемым с использованием того же доменного имени, либо идентичность ресурса должна измениться. В идеале идентичность никогда не должна меняться. Она должна быть неизменной. Но в интернете идентичность ресурса меняется каждый раз, когда сервер отвечает постоянным перенаправлением 301 или 308. Клиент должен обновить свою ссылку на этот ресурс и использовать новую идентичность с этого момента. К сожалению, старый идентификатор должен оставаться доступным для обслуживания откликов 301 или 308, поскольку нет способа узнать, когда все клиенты обновят свои ссылки. Клиенты должны связаться с удаленным сервером, чтобы узнать каноническую форму URL.

## Идентификация, не зависящая от местоположения

Мы рассмотрели лишь пару способов, с помощью которых идентификация объектов в приложении часто связана с их местоположением. Когда идентификация основана на автоинкрементном ID, этот ID имеет значение только в определенном месте и может быть сгенерирован только там. Когда идентификация основана на URL-адресах, местоположение узла, который отвечает на последующие команды, указывается прямо в идентификаторе. Когда идентификация зависит от местоположения, объекты проявляют определенную привязанность к месту своего происхождения. Приложения начинают испытывать проблемы с использованием этих объектов, когда их местоположение становится недоступным.

Окончательным решением каждой из этих проблем является идентификация объектов без зависимости от их местоположения. Независимая от местоположения идентификация обладает тремя полезными свойствами:

- она может быть сгенерирована из любого узла;
- она неизменяема;
- ее можно сравнивать.

Генерирование уникального идентификатора с любого узла решает проблему задержки при удаленных вставках. Будь то географически удаленный центр обработки данных аварийного восстановления или мобильное устройство в медленной сети, узел, способный генерировать собственные идентификаторы, может работать намного быстрее. Неизменяемые идентификаторы решают проблему сохранения старых доменов адресованными неопределенно долгое время. А сравнение между идентификаторами позволяет клиентам знать, когда они говорят об одном и том же объекте. Если бы им приходилось связаться с исходным местоположением, чтобы узнать каноническую форму идентификатора перед сравнением, они не смогли бы завершить свою транзакцию в изоляции.

Немного подумав, мы можем придумать идентификаторы, удовлетворяющие этим трем условиям. Такие идентификаторы не зависят от местоположения и поддерживают непрерывную работу изолированных узлов. Ниже приведены лишь несколько примеров.

### Естественные ключи

Вероятно, лучшим примером независимой от местоположения идентификации и тем, что должно быть по умолчанию в любом приложении, является естественный ключ. Изучите область, которую вы моделируете в своем приложении. Есть ли у нее уже атрибут, который уникально идентифицирует концепции в этой области? Является ли этот атрибут неизменяемым? Если да, рассмотрите возможность использования этого атрибута в качестве естественного ключа в модели.

Если вы создаете приложение для составления расписания занятий и вам нужно идентифицировать комнаты, посмотрите, не пронумерованы ли уже комнаты. Эти номера – хорошие кандидаты для естественных ключей в вашей системе. Номера комнат могут меняться со временем, но приложение для составления расписания уже учитывает время. Новый номер комнаты означает новую комнату, но прошлые события уже происходили в старой комнате. Приложению безразлично, что старая комната находилась в том же физическом пространстве.

Приложения, управляющие статьями, историями или вопросами, часто присваивают им теги. Хорошим естественным ключом является каноническое имя тега на основном языке (например, английском). Имя может быть канонизировано путем преобразования всех букв в строчные, опуская знаки препинания и заменяя пробелы дефисами, чтобы сделать его более дружественным к URL-адресу. Для преобразования тега fermats-last-theorem в полную фразу «Последняя теорема Ферма» или для его перевода на другие языки потребуется сопоставление. Однако естественный ключ легче сгенерировать на любом компьютере, чем синтетический идентификатор.

Некоторые естественные ключи являются первичными ключами, сгенерированными внешней системой. Если вы интегрируетесь с налоговой системой США, вы, вероятно, будете идентифицировать людей и компании по их налоговому идентификатору. Если вы получаете счет-фактуру от поставщика, хорошим естественным ключом для этого объекта будет номер счета, предоставленный поставщиком. Обычно нет веских причин для генерирования нового идентификатора, когда система на другом конце интеграции уже предоставила его.

## GUID

Когда естественный ключ недоступен, у нас есть механизмы для генерации идентификаторов, которые не совпадают на разных компьютерах. Это универсальные уникальные идентификаторы (UUID). Или если, как я, вы пришли к ним через компанию Microsoft, то это глобальные уникальные идентификаторы (GUID). Независимо от того, называете вы его UUID или GUID, это 128-битное число, представленное в шестнадцатеричном виде, в формате с переносом через дефис, который узнаваем для большинства разработчиков.

Изначально GUID генерировались с использованием MAC-адреса создающего компьютера и метки времени. Затем, когда генерация GUID стала более частой, метка времени была заменена счетчиком. Наконец, было признано, что случайные GUID, вероятно, так же хороши.

GUID предназначены для глобальной уникальности, но известны случаи коллизий. Хотя некоторые системы используют GUID для представления каждой строки в базе данных, моя практика была немного более сдержанной. Я генерирую GUID только для наиболее редко создаваемых объектов на самом высоком уровне, и то только в том случае, если естественные ключи нецелесообразны.

## Временные метки

Одним из самых простых способов идентификации объекта, созданного пользователем, является использование времени, когда пользователь его создал. Это хорошо работает, особенно когда в работе участвует только один человек. Детализация временных меток должна быть меньше секунды, чтобы гарантировать, что даже самые быстрые из действий человека получат уникальное значение. Миллисекунда в качестве дискрета является разумной и часто достижимой.

Хотя очень заманчиво сравнить временные метки, чтобы определить, какое событие произошло раньше другого, этого следует избегать. Временные метки возрастают только в пределах одного компьютера. И часы компьютера могут быть скорректированы вперед или назад. Такие корректировки, как переход на летнее время и пересечение часовых поясов, не являются проблемой – временные метки всегда должны фиксироваться в UTC. Но небольшие корректировки для исправления дрейфа часов должны быть разрешены.

Одной временной метки недостаточно для идентификации объектов в системе с большим числом пользователей. Она должна использоваться только в сочетании с другими формами идентификации.

## Кортежи

Использования только одного идентификатора, например временной метки, часто недостаточно, чтобы избежать коллизий. Но объедините различные формы идентификации вместе, и комбинация будет сильнее, чем любая из ее частей. Кортеж – это упорядоченный список значений, в котором каждый элемент имеет свой тип и значение.

Кортежи нередко записываются в виде списка в круглых скобках: (that-conference, day-2, 10:00, 136). Но не менее правильно записать кортеж в виде

пути: /that-conference/day-2/1000/136. Это придает им иерархическую структуру, что делает их пригодными для использования в URL (URL-адреса могут применяться в приложениях, но не в качестве идентификаторов объектов). Иерархия подразумевает, что объект имеет только одного владельца, который идентифицируется кортежем, имеющим на один элемент меньше. В предыдущем примере сессия, проводимая в комнате 136, принадлежит временному интервалу 10:00 в день 2.

Прозрачная природа кортежей делает их восприимчивыми к человеческой интерпретации. Это одновременно и преимущество, и недостаток. Хотя часто полезно видеть подразумеваемые отношения между объектами только по их идентификаторам, это иногда может вызвать путаницу. В некоторых случаях строгой иерархии не существует, но кортеж подразумевает ее своим выбором значений и порядка. А в других случаях значения в кортеже представляют собой изменяемые понятия. Мы можем выбрать либо изменить эти значения, и таким образом изменить идентичность объекта, либо сохранить старые значения и рисковать запутаться.

### Хеши

Чтобы избежать путаницы, вызванной прозрачной структурой данных, такой как кортеж, мы можем вместо этого выбрать непрозрачную структуру, например хеш. Хеш-функция принимает кортеж в качестве входных данных и выдает значение. Функция детерминирована – один и тот же кортеж всегда будет выдавать один и тот же хеш. Но в идеале функция должна быть непредсказуемой – должно быть трудно найти кортеж, который выдает заданный хеш.

Хеши имеют дополнительные преимущества перед исходными кортежами. Если кортеж содержит элементы переменной длины, например строки, хеши всегда имеют одинаковый размер. Кроме того, в то время как кортежи имеют тенденцию объединять данные, хеши имеют тенденцию разделять их. И если кортежи можно легко перепрограммировать, то хеши – только одним способом. Это делает их более подходящими для проблем, требующих определенной степени безопасности.

Многие системы, использующие хеши для идентификации, делают это по одной из этих причин. Блокчейн использует хеши для идентификации транзакций, чтобы их содержимое нельзя было легко изменить. Изменение одного элемента транзакции, например отправителя, получателя или суммы, изменит хеш. А поиск другой транзакции, которая производит тот же хеш, – неразрешимая проблема.

Git использует хеши для идентификации коммитов. Он делает это не для обеспечения их безопасности. Вместо этого, поскольку Git основан на файловой системе, наличие идентификатора постоянного размера помогает коммитам вписываться в имена и структуры данных. Кортеж, с которого он начинается, включает имя и электронную почту автора (естественные ключи), различия между двумя версиями и временную метку (до второй версии). Этот исходный кортеж имеет переменную длину и может быть довольно большим для значительных различий. Однако результирующий хеш равен 256 битам, или 64 шестнадцатеричным цифрам.

### Открытые ключи

В продолжение темы безопасности: открытые ключи являются отличным способом идентификации владельцев, например частных лиц или корпораций. Открытые ключи часто используются для цифровой подписи сообщений, доказывая их подлинность. Только человек, имеющий доступ к закрытому ключу, может создать подпись.

Сертификат – это полностью проверенный идентификатор владельца, часто включающий его имя, физическое местонахождение или юрисдикцию, а также личность проверяющей стороны. Сертификаты образуют свою собственную иерархию, поскольку личность стороны, подписавшей сертификат, предоставляется в виде открытого ключа.

Системы блокчейн используют открытый ключ в качестве единственного средства идентификации стороны. Каждая транзакция фиксирует отправителя и получателя по их открытым ключам. Чтобы заплатить кому-то биткойнами, достаточно знать его открытый ключ. Этого хватит, чтобы однозначно идентифицировать его для любого узла в распределенной сети.

### Случайные числа

Когда другие формы идентификации недоступны, приложение всегда может прибегнуть к помощи случайных чисел. Открытые ключи – это не что иное, как два случайных числа, которые были проверены на простоту, а затем перемножены. Современные идентификаторы GUID часто генерируются совершенно случайным образом, а не на основе MAC-адреса или временной метки. Поэтому правильным выбором будет просто использовать случайные числа напрямую, если они достаточно большие и достаточно случайны.

Как и временные метки, случайные числа никогда не должны использоваться в качестве единственной формы идентификации. Их следует комбинировать с другими идентификаторами для создания кортежа. Поскольку случайное число непригодно для интерпретации человеком, создание хеша этого кортежа часто является следующим шагом. В криптографии случайное число, добавленное к кортежу перед хешированием, называется «nonce» – сокращение от «number used once» (число, используемое один раз). В данном случае мы используем nonce, чтобы отличить объект от других, имеющих одинаковые значения кортежей.

При выборе генератора случайных чисел лучше всего придерживаться криптографически сильного алгоритма. Алгоритмы, используемые для генерации открытых ключей, общих секретов и nonce, специально выбираются для получения непредсказуемых результатов. Хотя вы, скорее всего, не будете полагаться на эти генераторы случайных чисел для защиты данных, вы будете использовать их результаты как часть идентификатора объекта. Если два узла используют один и тот же генератор случайных чисел, то вероятность коллизии высока.

Выберите наиболее подходящий механизм для генерации уникальных идентификаторов объектов. Какой бы метод вы ни выбрали, избегайте всего,

что может привязать идентификатор объекта к месту его генерации. Вместо этого выберите генератор, который отвечает следующим критериям:

- любой узел должен быть одинаково способен генерировать идентификаторы без обращения к центральной базе данных;
- идентификатор должен быть неизменяемым;
- одноранговые узлы должны иметь возможность сравнивать идентификаторы, чтобы знать, когда они говорят об одном и том же объекте.

Идентификация – это первый шаг к независимости от местоположения. Следующим шагом является обеспечение последовательного поведения без учета местоположения.

# Причинность

Когда мы начинаем рассуждать о поведении распределенной системы, мы пытаемся построить цепочку событий. Наша цель – предсказать, что произойдет на некотором удаленном узле когда-то в будущем. Чтобы получить такое предсказание, необходимо проанализировать эффект, который могут вызвать локальные действия.

Причинность сама по себе является трудноизмеримой категорией. Вы можете сказать, что опрокидывание одного домино вызвало падение следующего. Но упало бы второе само по себе? Мы хотели бы сказать наверняка, что нет. Однако, как известно любому, кто строил большую цепь домино, это трудно утверждать.

Причины многих событий в распределенной системе могут быть такими же сложными и непостижимыми, как цепь домино. И все же мы по-прежнему желаем некоторой предсказуемости от системы. Поэтому мы должны найти разумный заменитель причинности, который было бы легче измерить и использовать для прогнозирования.

Хотя мы не всегда можем с уверенностью сказать, что одно событие вызвало другое, мы можем сказать, что причина произошла раньше следствия. На момент написания этой книги путешествия во времени все еще невозможны. Возможно, достаточно сказать «произошло раньше». Вероятно, достаточно использовать порядок событий для определения причинности. Давайте применим это представление причинности к шагам в программе и сравним это с нашей интуицией.

## Упорядочивание шагов

Мы часто думаем о программе как о последовательности шагов. Шаги выполняются по порядку по мере выполнения программы. Легко посмотреть на два шага, выполняющихся в одной и той же программе, как, например, на рис. 4.1, и сказать, что один из них произошел раньше другого.

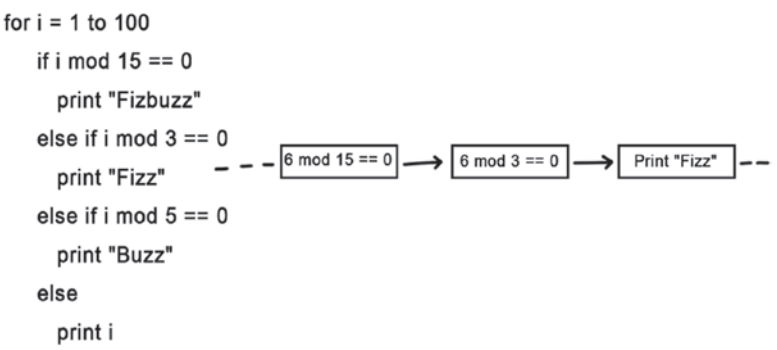


**Рис. 4.1.** Шаги процесса

Использование порядка шагов в качестве замены причинности приводит к тому, что мы говорим, что один шаг программы вызывает следующий. В некотором смысле это верно. Программа выполняется последовательно, поэтому разумно сказать, что выполнение одного шага приведет к тому, что компьютер выполнит следующий. Даже если эти два шага оперируют с разными объектами и не зависят друг от друга, они, по крайней мере, связаны во времени. Программа не дойдет до второго шага, не выполнив первый.

Вы можете подумать, что оператор `goto`, который переходит ко второму шагу, нарушает это понятие причинности. Программа выполняет второй шаг, не выполнив первый. Однако в этой ситуации мы бы заметили, что оператор `goto` выполнится раньше, чем второй шаг. Для нас интересен не порядок появления шагов в коде, а порядок, в котором они происходят во времени. И поэтому именно `goto` вызвало выполнение шага. Это согласуется с нашей интуицией относительно оператора, который обеспечит переход к другому шагу. «Случилось раньше» выглядит как хорошая мера причинности.

Когда мы пытаемся обобщить шаги в одной программе на многопоточные или многопроцессные системы, все становится немного сложнее. Мы не можем четко сказать, какой из двух шагов, выполняющихся в разных процессах, произошел раньше другого. Процессы могут выполняться в параллельных потоках или даже на разных компьютерах. Не существует единых часов, которые могут помочь нам расположить эти шаги по порядку.

Однако мы можем заметить, что два процесса, выполняющихся независимо друг от друга, не вызывают никаких изменений в поведении друг друга. Они не имеют причинно-следственной связи. До тех пор, пока они не взаимодействуют, ничто, происходящее в одном из них, не может повлиять на другой.

Когда же они *общаются*, причинно-следственная связь четко прослеживается. Если один процесс посылает сообщение, а другой процесс получает его, то мы знаем, что шаг *отправки* произошел до шага *получения*. И отправка сообщения реально вызвала его получение. Имея на руках этот факт, мы можем начать причинно упорядочивать шаги, которые произошли в различных процессах. Именно так Лесли Лэмпорт (Leslie Lamport) определил порядок событий в своей работе 1978 года о распределенных системах[1].

[1] *Leslie Lamport.* Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21 (7): 558–565, July 1978.

## Транзитивное свойство

Отношение, в котором один шаг произошел раньше другого, обладает еще одним полезным свойством – оно является транзитивным. То есть если один шаг произошел перед вторым, а второй шаг произошел перед третьим, то мы знаем, что первый шаг произошел перед третьим. Это легко увидеть, когда все шаги находятся в одном и том же процессе. Эти шаги идут последовательно. Но транзитивное свойство действует и тогда, когда мы пересекаем границы процесса.

Возьмем, к примеру, веб-браузер. Пользователь дает команду браузеру перейти на заданный URL. Браузер только что выполнил два шага: получил URL от пользователя и отправил запрос на веб-сервер. Поскольку эти шаги произошли в одном и том же процессе, мы знаем, что один из них произошел раньше другого – ввод URL пользователем произошел раньше, чем отправка запроса.

Теперь давайте посмотрим, что происходит на веб-сервере. Когда он получает запрос, он загрузит запрашиваемый файл с жесткого диска. Получение запроса и загрузка файла – это два этапа одного и того же процесса. Поэтому мы можем сказать, что получение произошло до загрузки.

Поскольку эти два процесса общаются друг с другом, передавая сообщения, мы можем также упорядочить некоторые шаги между процессами. Мы можем с уверенностью сказать, что отправка запроса произошла раньше, чем его получение, даже если они произошли в разных процессах. Как показано на рис. 4.2, свойство транзитивности позволяет нам связать эти события в цепочку. Мы можем точно сказать, что пользовательский ввод произошел до загрузки файла.

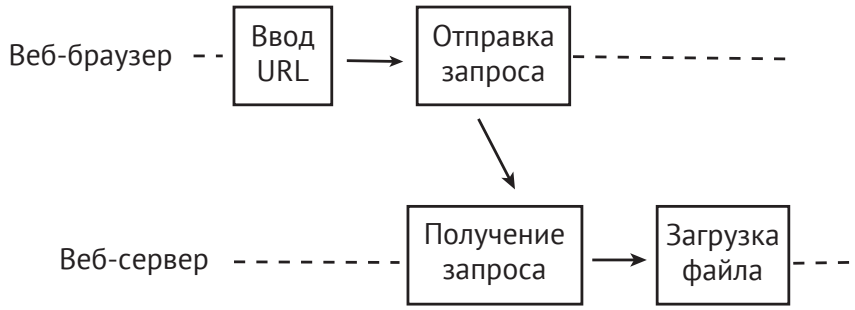


**Рис. 4.2.** Порядок шагов в двух процессах

Поскольку мы используем «произошло раньше» в качестве обозначения причинно-следственной связи, то на самом деле мы утверждаем, что пользовательский ввод *вызвал* загрузку файла. Это хорошо согласуется с нашей интуицией. Мы можем представить, что пользователь хотел, чтобы веб-сервер загрузил файл, и таким образом эта причинно-следственная цепь событий служит для реализации намерения пользователя. Мы также можем предположить, что веб-сервер, вероятно, не загрузил бы этот конкретный файл, если бы пользователь не ввел URL. Но о намерениях и «могло бы быть» трудно рассуждать. «Случилось раньше», однако, очень ясно.

Мы можем четко сказать, когда знаем, что один шаг произошел раньше другого. Мы также можем четко сказать, когда понятия не имеем.

## Параллелизм

Хотя в нашем предыдущем примере можно было точно сказать, что пользовательский ввод произошел до загрузки файла, это не всегда так. Некоторые шаги, происходящие в разных процессах, не так легко упорядочить, даже используя свойство транзитивности.

Возьмем, к примеру, шаг, на котором веб-сервер открывает сокет. Как показано на рис. 4.3, «открыть сокет» происходит до «получить запрос» и выполняется в одном и том же процессе. И как мы видели в предыдущем анализе, «ввод URL» также происходит до «получения запроса». Но свойство транзитивности не позволяет нам сказать, какая из операций – «открыть сокет» и «ввод URL» – произошла раньше другой. Они обе произошли раньше «получить запрос», но это ничего не говорит об их относительном порядке. Лэмпорт назвал два шага, которые нельзя расположить по порядку, *параллельными*.

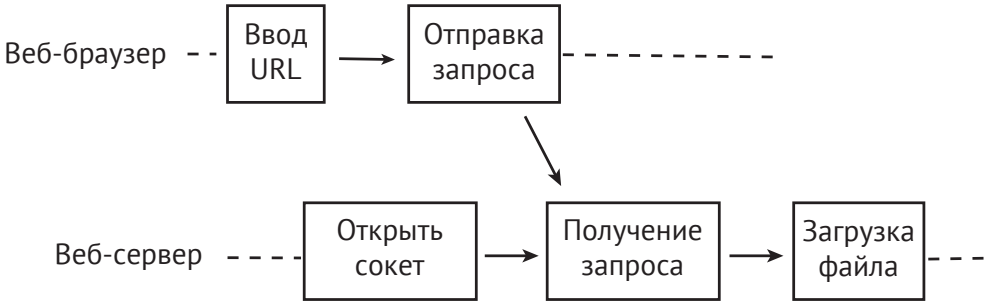


**Рис. 4.3.** Нет причинно-следственной связи между «открыть сокет» и «ввести URL»

Это определение параллельности немного отличается от других, которые вы могли слышать. Параллельные операции в многопоточной системе могут выполняться параллельно. Вы чувствуете, что если два события одновременны, то они происходят в одно и то же время.

По определению Лэмпорта, мы не знаем, действительно ли два события произошли одновременно. Они могли быть разделены большим промежутком фактического времени на физических часах. Например, может оказаться, что веб-сервер открыл сокет за несколько часов до того, как пользователь ввел URL. На самом деле это вполне вероятно. Но то, что может быть и что вероятно, не имеет значения в этом разговоре. Именно тот факт, что мы *не можем этого знать*, делает эти два события параллельными.

Параллелизм – это то, что делает распределенные системы такими сложными для осмысления. Если бы не было параллельных шагов, мы могли бы упорядочить все шаги. Если каждый шаг может быть упорядочен относительно другого шага, то в итоге мы получим полностью упорядоченную последовательность. Думать о такой системе было бы намного проще, потому что она всегда ведет себя так, как будто вся сеть работает на одном компьютере.

Хотя о полностью упорядоченной системе было бы легче думать, она не обладала бы теми свойствами, которые мы хотим видеть в распределенной системе. Она не будет масштабироваться по мере увеличения количества аппаратного обеспечения, поскольку полностью упорядоченные шаги не могут выполняться параллельно. Она не сможет автономно обслуживать клиентов в разных местах, потому что шаги, которые программа выполняет для обслуживания одного клиента, должны быть упорядочены с другими в реальном времени. И это не позволит автономную работу, поскольку шаги, выполняемые на автономном компьютере, будут идти не по порядку с остальной частью сети. Таким образом, параллелизм является одновременно героем и злодеем этой истории.

### Частичный порядок

Если бы вы сравнили два любых шага, выполняемых в одном и том же процессе, вы могли бы сказать, какой из них был первым. Эти шаги *полностью упорядочены*. Они происходят последовательно.

Однако если вы сравните два шага, выполняющихся в разных процессах, вы, возможно, сможете определить, какой из них был первым. Если один из них предшествовал отправке сообщения, получение которого предшествовало второму, то транзитивное свойство говорит нам, что первый шаг произошел раньше второго. Но если это не так – если два шага происходят параллельно, – тогда вы не можете сказать, что было первым. Поскольку иногда вы можете сказать, а иногда нет, то выполнение шагов в многопроцессной системе называется *частично упорядоченным*.

Так как мы используем «произошло раньше» в качестве обозначения причинности, то можем сказать, что причинность сама по себе частично упорядочена. Некоторые вещи причинно-следственно связаны – мы можем четко сказать, что является причиной, а что – следствием. Ввод пользователем URL-адреса в браузере вызвал загрузку файла в веб-сервере. Но некоторые вещи не имеют причинно-следственной связи. Открытие веб-сервером сокета не заставило пользователя ввести URL, равно как и ввод URL не заставил веб-сервер открыть сокет.

Частичный порядок накладывает на систему меньше ограничений, чем полный порядок. Он позволяет выполнять некоторые шаги параллельно. Он позволяет устройствам действовать автономно при отключении от сети. Он дает узлам возможность действовать независимо без постоянной синхронизации. Признание того, что причинность частично упорядочена, дает нам мощный инструмент для анализа распределенных систем. Мы можем лучше понять их возможности, а также их ограничения. И мы можем сделать лучший выбор между ними.

## Теорема CAP

Вероятно, самой известной математической идеей во всех распределенных системах является теорема CAP. Она была постулирована Эриком Брюэром (Eric Brewer) на Симпозиуме по принципам распределенных вычислений

в 2000 году[1]. Формально доказана Сетом Гилбертом (Seth Gilbert) и Нэнси Линч (Nancy Lynch) в 2002 году[2]. Теорема CAP связывает идеи согласованности, доступности и устойчивости к разбиению (consistency, availability and partition tolerance). На нее часто ссылаются, говоря, что вы можете иметь только два из трех условий.

Согласованность означает разные вещи в разных контекстах. К сожалению, как это обычно бывает, у этого понятия нет четкого определения. Например, если вы знакомы с реляционными базами данных, то, вероятно, впервые услышали о согласованности в связи с ACID-свойствами транзакции – атомарностью, согласованностью, изолированностью и долговечностью (atomic, consistent, isolated and durable). Атомарность легко определяется как «все или ничего». Изолированность просто означает, что параллельные транзакции не влияют друг на друга. А долговечность означает, что изменения сохраняются.

Но согласованность в данном контексте не так просто определить. Рабочее определение заключается в том, что согласованная транзакция – это транзакция, которая не нарушает никаких инвариантов. Она «фиксирует только законные результаты»[3]. Проблема в том, что инварианты, определяющие законный результат, поступают из двух источников – системы управления базой данных и приложения. Инварианты системы управления базой данных включают такие гарантии, как «первичные ключи уникальны» и «внешние ключи ссылаются на существующие строки». Определяемые приложением инварианты, если они вообще существуют, определяются в терминах проблемной области, например «все балансы равны нулю или положительны». Если бы мы говорили только о четко определенных гарантиях, обычно используемых в системах управления базами данных, у нас был бы некоторый шанс доказать некоторые общеприменимые теоремы. Но со всеми возможностями выбора, который приложение может сделать при определении своих инвариантов, специфичных для конкретной области, нам очень трудно написать содержательное доказательство. Поэтому мы будем использовать более точное определение.

## Определение CAP

Определение **согласованности**, которое использует теорема CAP, относится именно к узлам распределенной системы. Оно гласит, что если я запрошу значение у двух разных узлов, то они дадут мне одинаковый ответ, как показано на рис. 4.4. Если узлы согласованы, то их ответы будут совпадать. Если ответы не совпадают, значит, узлы не согласованы.

---

[1] *E. Brewer*. Towards Robust Distributed Systems. Proc. 19th Ann. ACM Symposium on the Principles of Distributed Computing (PODC 00), ACM, 2000, p. 7–10.

[2] *Seth Gilbert and Nancy Lynch*. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, Volume 33 Issue 2 (2002), p. 51–59.

[3] *Haerder T., Reuter A*. December 1983. Principles of Transaction-Oriented Database Recovery // Computing Surveys. 15 (4): 287–317.

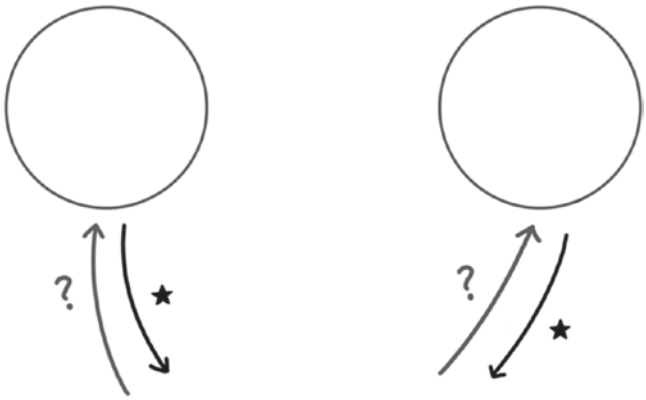


**Рис. 4.4.** Согласованность

Это сильно отличается от определения, используемого в ACID. На самом деле можно утверждать, что это ближе к атомарности, чем к согласованности. Либо оба узла имеют последнюю версию значения, либо ни один из них. Но там, где атомарность – и вообще каждая из гарантий ACID – касается изменений в одной базе данных, согласованность в CAP касается узлов в распределенной системе.

Продолжая далее, буква A в CAP означает **доступность**. Узел доступен, если он отвечает в течение разумного количества времени на любой запрос, как показано на рис. 4.5. В связи с этим возникает вопрос: «Что такое разумное количество времени?» Ответ на этот вопрос: «Время, необходимое для восстановления разбиения сети».

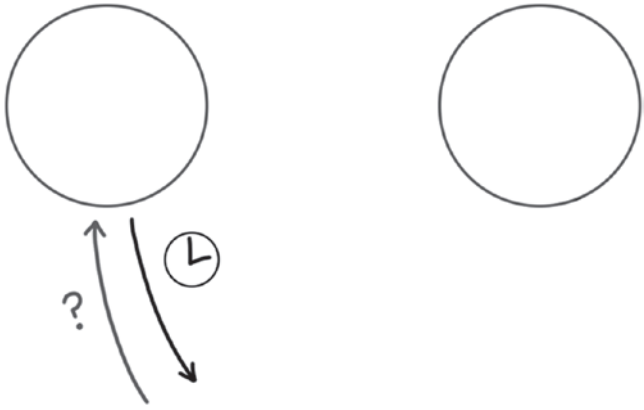


**Рис. 4.5.** Доступность

Итак, что такое разбиение сети? Это P в CAP: устойчивость к разбиению. Разбиение сети – это условие, которое препятствует прохождению сообщений в сети, как показано на рис. 4.6. Однако разбиение является лишь временным. Через некоторое время соединение будет восстановлено. Но в то же время устойчивость к разбиению обещает, что система будет продолжать функционировать.

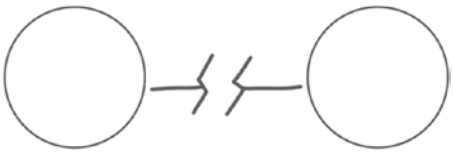

**Рис. 4.6.** Устойчивость к разбиению

Вооружившись этими определениями, мы можем наконец сформулировать утверждение Брюэра. Ни одна распределенная система, независимо от того, какой алгоритм она использует, не может одновременно гарантировать согласованность, доступность и устойчивость к разбиению в любой заданный интервал времени. Если в течение этого интервала сеть будет разделена, то система будет либо несогласованной, либо недоступной.

Это одна из тех восхитительных теорем, которые ставят перед вами задачу найти алгоритм, который работает, как Гёдель просит вас написать формулу, которая определяет, истинно ли другое выражение, или Тьюринг просит вас написать программу, которая определяет, завершается ли другая программа, или два генерала приказывают вам найти надежный способ, чтобы они могли общаться. Доказательство не должно угадывать, что вы можете придумать. Оно может просто продемонстрировать с помощью логики, что все, что вы придумали, не будет соответствовать задаче.

## Доказательство теоремы CAP

Представьте себе, что у вас есть система, состоящая из различных компьютеров, которые мы будем называть узлами. Каждый узел имеет свое собственное внутреннее состояние. Это состояние, однако, невидимо для нас. Единственное, что мы можем сделать в качестве стороннего наблюдателя, – это послать сообщения узлам и посмотреть, как они отвечают.

Сообщение, которое мы пошлем узлам, будет *чтением* (`read`) и *обновлением* (`update`). Если мы пошлем узлу сообщение на *чтение*, он отправит нам в ответ значение. Если мы пошлем ему *обновление*, он предположительно запишет значение, а затем ответит подтверждением. Я говорю «предположительно», потому что мы не можем увидеть его внутреннее состояние.

Единственный способ наблюдать за узлом – посылать ему сообщения. А сообщения имеют следующий механизм: если обновить узел в изоляции, а затем, после подтверждения, послать ему сообщение `read`, он вернет значение, которое я только что обновил, как показано на рис. 4.7. Узел действует так, как будто он сохраняет состояние, чтобы мы могли получить его позже.

Здесь будьте осторожны. Мы не можем точно сказать, что делают эти узлы. Их внутренние действия остаются неопределенными. Это важно для того, чтобы доказательство было общим. Если бы мы решили, что они действительно *хранят* внутреннее состояние, это ограничило бы типы алгоритмов, о которых мы могли бы делать утверждения.

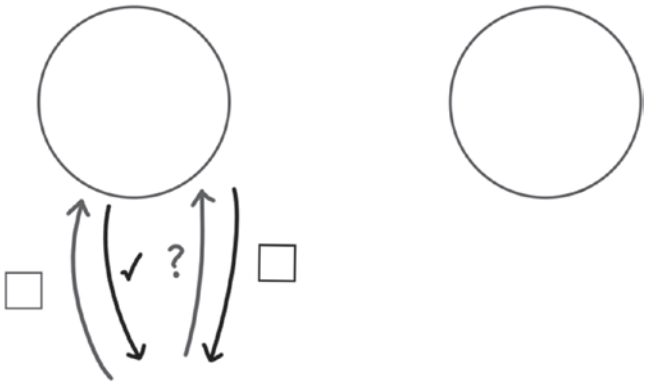


**Рис. 4.7.** Обновление и чтение

Аналогично сообщения `read` и `update` также не ограничивают наш выбор в алгоритме. Нам не нужно придумывать способ отправки сообщений о чтении и обновлении для достижения согласованности. На самом деле эти сообщения могут даже не использоваться алгоритмом. Они существуют только как способ создания теста.

## Проверка алгоритма

Итак, вот в чем заключается задача. Я прошу вас предоставить алгоритм. Вы можете разработать любой алгоритм, который вам нравится. Вы сами выбираете шаги. Вы выбираете структуры данных. Я загружу этот алгоритм в два узла. Они взаимодействуют друг с другом, передавая сообщения между собой. Вы выбираете, какие сообщения они будут использовать.

Затем я проведу тест. Я начну с наблюдения за тем, что узлы изначально согласованы. Я могу сказать, что они согласованы, посылая каждому из них сообщения `read` и наблюдая, что они возвращают один и тот же ответ.

Затем я пошлю одному из них *обновление* и подожду, пока он ответит подтверждением. Поскольку он подтвердил, я могу проверить выполнение теста, отправив тому же узлу сообщение `read`. Если он ведет себя правильно, он вернет то значение, которое я ему отправил. Примерно в половине тестов я проведу эту проверку. Я отвергну ваш алгоритм, если он не сможет выполнить тесты.

В другой половине тестов я собираюсь обратиться к его соседу для выполнения *чтения*, как показано на рис. 4.8. Если я получу то же значение, которое только что *обновил*, то система демонстрирует согласованность. Если и *обновление*, и *чтение* происходят в течение определенного интервала времени, то система демонстрирует *доступность*. Чтобы быть полностью честным, я даже позволю вам сказать мне, каким должен быть этот интервал.

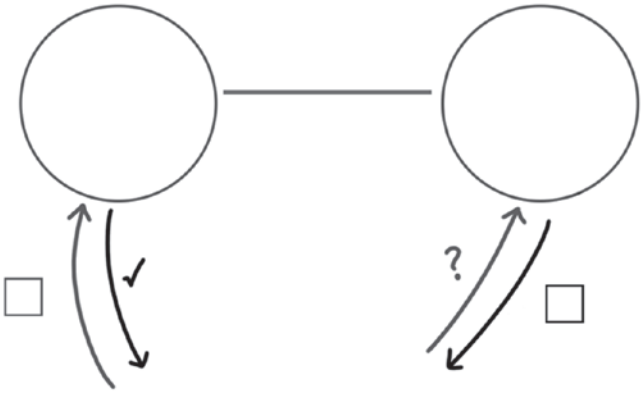


**Рис. 4.8.** Проверка алгоритма

Но здесь я использую хитрость. Во время тестирования я создам разбиение сети. В течение этого интервала два узла не смогут общаться друг с другом. Разбиение будет длиться чуть дольше, чем заданная вами продолжительность. Хотя связь в конечном итоге будет восстановлена, это не будет достаточно быстро для того, чтобы алгоритм продемонстрировал доступность и остался согласованным. Поэтому алгоритму придется выбрать один из трех вариантов поведения, показанных на рис. 4.9.

1. Первый узел может заблокироваться во время *обновления*, пока он не сможет передать значение и затем подтвердить результат. Если это так, то *обновление* занимает больше времени, чем заданный интервал, и поэтому система недоступна.

2. Второй узел может заблокироваться во время *чтения*, пока он не получит значение от первого. Если это так, то *чтение* занимает больше времени, чем заданный интервал, и поэтому система снова будет недоступна.

3. Система может решить вернуть значение до истечения интервала. Если это так, то не было никакого способа, чтобы *обновленное* мною значение смогло распространиться на второй узел, чтобы быть *прочитанным*. Второй узел не может вернуть то же значение, что и первый, и поэтому система не является согласованной.

Во время этого интервала разбиения сети система не может быть одновременно согласованной и доступной. И поэтому кажется, что мы обречены на выбор.

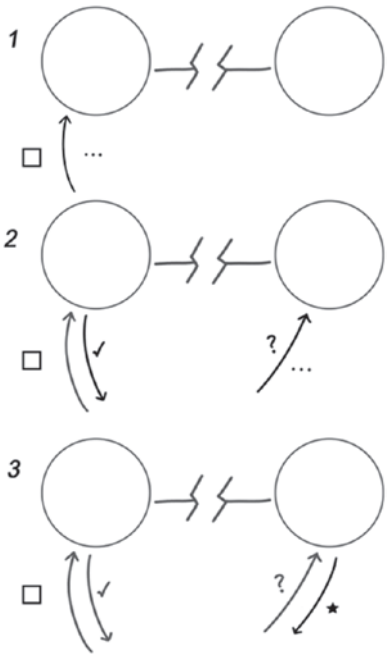


**Рис. 4.9.** Три возможных варианта поведения при разбиении сети

## Конечная согласованность

Если мы не можем ожидать, что различные узлы в распределенной системе будут иметь одинаковое состояние, тогда на что мы можем надеяться? Как мы можем выполнить какую-либо работу, если получаем разные ответы от узлов, к которым обращаемся?

Согласованность в любой момент времени может быть недостижима, но надежда не потеряна. Мы можем достичь согласованности, если будем ждать достаточно долго. В конце концов, узлы придут к согласию друг с другом. Это понятие называется конечной согласованностью[1].

Хотя было бы желательно требовать согласованности в любой момент времени, это может оказаться непрактичным. Если мы ослабим наши ограничения, то обнаружим, что можем достичь гораздо более приемлемого компромисса. Вместо того чтобы настаивать на согласованности в каждый данный момент, возможно, мы можем терпимо относиться к меньшей степени согласованности. Разговор должен стать более детальным.

[1] *Terry D. B., Theimer M. M., Petersen K., Demers A. J., Spreitzer M. J., Hauser C. H.* (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. Proceedings of the fifteenth ACM symposium on Operating systems principles – SOSP '95. P. 172.

## Виды согласованности

Марк Шапиро (Marc Shapiro), научный сотрудник французского Национального института информатики и управления (French National Institute for Computer Science and Control Science – Inria), и Нуно Прегиса (Nuno Preguiça), доцент факультета наук и технологий университета da Universidade (Faculdade de Ciências e Tecnologia da Universidade), стремились понять компромиссы в отношении согласованности на формальном уровне. Каждый из них разработал специальные решения для достижения конечной согласованности, включая Treedoc, реплицированную структуру данных для совместного редактирования текста[1]. Каждый из этих проектов требовал собственного формального доказательства. Они хотели получить более общий результат.

Основываясь на своих предыдущих результатах, Шапиро и Прегиса вместе со своими коллегами определили три различных вида согласованности[2]. Различия между ними приводят к общему результату, который они искали. Они переопределили вид согласованности, используемый в теореме CAP, как **сильную согласованность**. Это гарантия того, что все узлы будут сообщать, что находятся в одном и том же состоянии в любой момент времени. С другой стороны, они использовали термин **конечная согласованность**, что означает, что узлы в конечном итоге достигнут одного и того же состояния, пока они могут продолжать общаться друг с другом. Это может потребовать некоторого дополнительного алгоритма достижения согласия, например разрешения конфликтов.

Зависимость от алгоритмов достижения согласия влечет за собой значительные накладные расходы. Узлам может потребоваться избрать ведущего для принятия окончательного решения, что создает узкое место. Или они могут запустить сложный алгоритм типа Paxos, чтобы определить большинством голосов, каким должно быть окончательное состояние. По этим причинам Шапиро и Прегиса решили выделить третий вид согласованности. **Сильная конечная согласованность** обещает, что все узлы достигнут одного и того же состояния в тот момент, когда они все получат одинаковые обновления. Узлам не нужно общаться между собой, чтобы достичь согласия и разрешения конфликтов.

Теорема CAP показала нам, что сильная согласованность несовместима с доступностью. Разрешение алгоритма достижения согласия означает, что конечная согласованность может повлечь за собой некоторые нежелательные накладные расходы. Поэтому мы, как и Шапиро и Прегиса, сосредоточим свое внимание на сильной конечной согласованности (strong eventual consistency – SEC).

---

[1] *Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, Mihai Leția*. A commutative replicated data type for cooperative editing. 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), Jun 2009, Montreal, Québec, Canada. P. 395–403, ff10.1109/ ICDCS.2009.20ff. ffinria00445975.

[2] *Shapiro Marc, Preguiça Nuno, Baquero Carlos, Zawirski Marek.* Conflict-free Replicated Data Types. Institut National de Recherche en Informatique et en Automatique, No 7687, 2011.

## Сильная конечная согласованность в системе ретрансляции

Поскольку SEC является нашей заявленной целью, давайте построим полезный пример. Построим распределенную систему, основанную на передаче сообщений, и посмотрим, какими свойствами она должна обладать, чтобы удовлетворять SEC.

Эта распределенная система состоит из узлов, соединенных в некоторую сеть. Сеть является связной, то есть если сеть не разделена на части (что будет происходить время от времени), существует путь от любого узла к любому другому узлу. Эти пути не должны быть прямыми, они могут проходить через любое количество промежуточных узлов.

Некоторые узлы получают новую информацию извне сети. Когда они это делают, они формулируют сообщение, которое сами обрабатывают, а затем отправляют по сети соседним узлам. Когда сосед получает сообщение, он обрабатывает его и передает своим соседям. Каждый узел выполняет некий алгоритм, чтобы определить, когда переслать сообщение и кому. Этот алгоритм гарантирует, что в конечном итоге каждый узел получит каждое сообщение.

Важно отметить, что мы *не* требуем явно, чтобы каждый узел получил каждое сообщение ровно один раз. Мы также не требуем, чтобы каждый узел получал сообщения в одном и том же порядке. Какой бы алгоритм пересылки мы ни придумали, он должен только обеспечить конечную доставку.

Теперь рассмотрим внутреннее состояние узла в распределенной системе. По мере того как узел обрабатывает сообщение, он переходит из одного состояния в другое. Сообщение можно рассматривать как функцию, принимающую начальное состояние в качестве входа и производящую результирующее состояние в качестве выхода. Система имеет сильную конечную согласованность (SEC), если после просмотра всех сообщений все узлы приходят в одно и то же состояние. Мы можем определить, какими свойствами должны обладать эти функции, чтобы достичь SEC.

Во-первых, ответ каждого узла на каждое сообщение должен быть **идемпотентным**. Если узел видит одно и то же сообщение дважды подряд, то он должен оказаться в том же состоянии, как если бы он видел его только один раз. Во-вторых, реакция каждого узла на каждую пару сообщений должна быть **коммутативной**. Если узел видит два сообщения в одном порядке, он должен оказаться в таком же состоянии, как если бы он видел их в противоположном порядке.

Вместе взятые, идемпотентность и коммутативность достаточны для доказательства SEC. Это так, потому что каждый узел в конечном итоге видит каждое сообщение хотя бы один раз. Этот результат справедлив только для распределенной системы, основанной на ретрансляции, которую мы определили. Он предполагает, что сообщения передаются точно такими, какие они есть, не фильтруются, не изменяются и не обобщаются. Мы найдем более общий результат в следующей главе, а пока давайте рассмотрим систему на основе ретрансляции.

## Идемпотентность и коммутативность

Матиас Верраес (Mathias Verraes) пошутил в Твиттере (рис. 4.10):

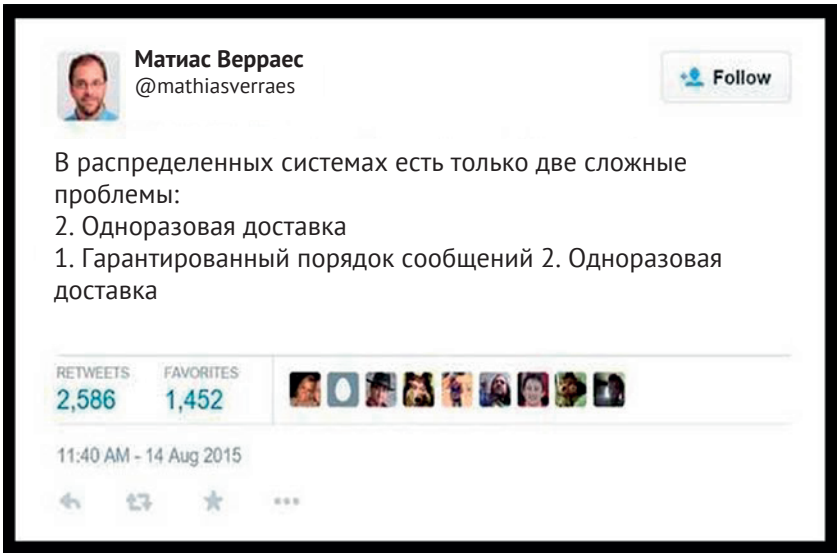


**Рис. 4.10.** https://twitter.com/mathiasverraes/status/632260618599403520

Как и все хорошие шутки, эта абсолютно верна. Трудно гарантировать, что сообщение будет доставлено ровно один раз – не потеряно и не продублировано. Еще труднее гарантировать, что сообщения придут в том порядке, в котором были отправлены.

Сетевые протоколы были изобретены специально для решения этих двух сложных проблем. Например, AMQP – это протокол обмена сообщениями, который может быть настроен для предоставления ряда гарантий. Его можно использовать как транспорт с наилучшими усилиями, при котором сообщение гарантированно будет получено не более одного раза. Он также может быть настроен на надежную доставку, которая гарантирует, что сообщение будет получено по крайней мере один раз, но, вероятно, более одного раза и, возможно, не по порядку. С немного большими накладными расходами он может выполнять антидупликацию, которая пытается гарантировать доставку ровно один раз. И, приложив геркулесовы усилия, он может сериализовать сообщения в канале, гарантируя, что они будут доставлены в том же порядке, в котором были отправлены, хотя вы не будете довольны производительностью.

Авторы инфраструктурных компонентов, которые полагаются на AMQP, таких как RabbitMQ, часто советуют потребителям допускать дублирование сообщений[1]. Стоимость запуска очереди сообщений с антидупликацией или сериализованными каналами может быть непомерно высокой. Вместо этого они рекомендуют заставить нижестоящие узлы терпимо относиться к сообще-

[1] [C]onsumer applications will need to perform deduplication or handle incoming messages in an idempotent manner. RabbitMQ Reliability Guide. https://www.rabbitmq.com/reliability.html.

ниям, которые приходят несколько раз или не по порядку. Именно это и означает идемпотентность и коммутативность.

Нижележащий узел, который допускает дублирование сообщений, является *идемпотентным*. Он будет оставаться в своем текущем состоянии после того, как увидит дублированное сообщение. Классическим примером идемпотентного узла является HTTP-сервер, получающий сообщение `put`. Сообщение несет желаемое состояние ресурса, заданного с помощью URI. Если он получает сообщение `put` во второй раз, HTTP-сервер просто снова установит желаемое состояние, как показано на рис. 4.11. Конечный результат такой же, как если бы HTTP-сервер получил только одно сообщение `PUT`.

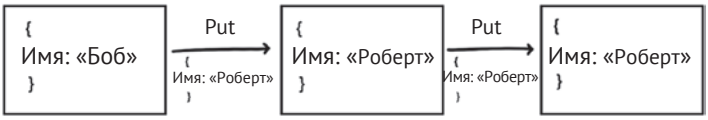


**Рис. 4.11.** Идемпотентный put

Нижележащий узел, который допускает сообщения не по порядку, называется *коммутативным*. Это происходит от математического свойства коммутативности, которое гласит, что оператор имеет один и тот же результат, независимо от того, в какой последовательности заданы его операнды. Коммутативное свойство сложения гласит, что `a+b = b+a`.Умножение также является коммутативным, а вычитание и деление – нет. В аналогичном смысле узел является коммутативным по отношению к двум сообщениям, если он оказывается в одном и том же состоянии независимо от того, какое сообщение видит первым.

## Получение сильной конечной согласованности

Узел может быть идемпотентным по отношению к набору сообщений, но не коммутативным. Например, HTTP-сервер, получающий два разных PUT-запроса на один и тот же ресурс, будет вести себя по-разному в зависимости от их последовательности. Ресурс окажется в состоянии, описанном последним сообщением, которое он видит. Измените порядок сообщений, и вы измените конечное состояние ресурса, как показано на рис. 4.12.

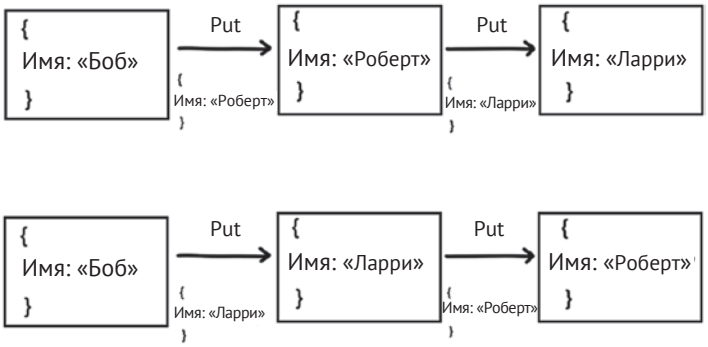


**Рис. 4.12.** Некоммутативный PUT

Сильная конечная согласованность требует идемпотентности и коммутативности. Давайте вернемся к нашему рабочему определению сильной конечной согласованности, чтобы понять, почему это так.

Распределенная система ретрансляции является SEC, если все узлы, увидев один и тот же набор сообщений хотя бы один раз в любом порядке, достигают одного и того же состояния. Конечно, все они должны начинать в одинаковом состоянии. Если бы набор сообщений был пустым, проблема была бы неинтересной – все узлы по-прежнему находились бы в начальном состоянии. А если набор содержал только одно сообщение, в конечном итоге согласованность будет зависеть лишь от идемпотентности. Узлы, получившие дубликаты копии этого сообщения, останутся в том же состоянии.

Таким образом, нам нужно тщательно рассмотреть только случай, когда набор содержит более одного сообщения. Давайте рассмотрим, как это может происходить. Если каждый узел получил каждое сообщение ровно один раз, тогда мы могли бы утверждать, основываясь только на коммутативности, что все они достигнут одного и того же состояния. Или если каждый узел получил каждое сообщение по порядку, но некоторые были удвоены или утроены, тогда мы могли бы утверждать, основываясь только на идемпотентности. Именно тот факт, что все может перепутаться, заставляет нас остановиться и рассмотреть все возможности.

Возьмем, к примеру, пару PUT-запросов к HTTP-серверу. Как мы уже отмечали ранее, PUT идемпотентен, но не коммутативен. Поэтому если HTTP-сервер увидит одно и то же сообщение PUT, продублированное немедленно, он не изменит своего состояния. Однако если между дубликатами есть промежуточные сообщения, то возникнет проблема. Если второе PUT было получено между первым и его дубликатом, как на рис. 4.13, то HTTP-сервер перезапишет свое изменение, когда придет дубликат. Для того чтобы вести себя согласованно, узел должен будет игнорировать дубликат вообще.

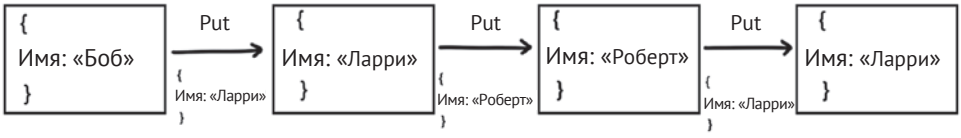


**Рис. 4.13.** Промежуточный PUT

Таким образом, для данного набора сообщений {`m1, m2, m3, ...`} сценарий, который мы должны тщательно рассмотреть, – это если `m1` дублируется после получения некоторого количества промежуточных сообщений. Мы хотели бы сказать, что

```
m1+m2+m3+m1 =
m1+m2+m3
```

Как оказалось, это действительно можно доказать для идемпотентного и коммутативного набора операций. Во-первых, мы наблюдаем, что дублированное сообщение (`m1`) коммутативно с сообщением, которое было получено непосредственно перед ним (`m3` в данном примере). Поэтому мы можем поменять их местами без изменения состояния узла. Итак:

```
m1+m2+m3+m1 =
m1+m2+m1+m3
```

Мы просто поменяли местами `m1` и `m3` в конце. Это перемещает дублирование сообщения на один шаг раньше в последовательности.

Но теперь мы замечаем, что можем снова использовать коммутативное свойство, на этот раз с `m2`. То есть

```
m1+m2+m1 =
m1+m1+m2
```

И таким образом дублирование переходит еще на один шаг раньше. Мы можем продолжать делать это до тех пор, пока дубликат сообщения не окажется рядом с оригиналом. В этот момент мы просто используем свойство идемпотентности дублированного сообщения, чтобы утверждать, что получить его дважды так же хорошо, как и один раз. Другими словами:

```
m1+m1 =
m1
```

Итак, мы показали, что поскольку узел является идемпотентным и коммутативным по отношению ко множеству сообщений, он достигнет того же состояния после того, как увидит одно из дублированных сообщений, независимо от того, сколько других сообщений вмешалось:

```
m1+m2+m3+m1 =
m1+m2+m3
```

И это обобщается на любое количество сообщений. Таким образом, проблема сводится к получению некоторого набора сообщений в любом порядке, но без дубликатов. Нам только нужно полагаться на коммутативность, чтобы гарантировать, что любая такая последовательность даст один и тот же результат.

И именно поэтому наш пример PUT не демонстрирует сильной конечной согласованности. Хотя он идемпотентен, он не коммутативен. Оба свойства необходимы для достижения SEC в распределенной системе на основе ретрансляции.

## Система управления контактами

Мы с другом создали систему управления контактами еще в те времена, когда персональные цифровые секретари (personal digital assistants – PDAs) подключались к рабочей станции через последовательный порт RS-232. В то время на современном уровне была система ActiveSync от Microsoft. Мы думали, что сможем создать лучший продукт.

Решение, которое мы придумали, представляло собой систему хранения и пересылки сообщений, где узлы (рабочие станции и PDA) обрабатывали сообщения в идемпотентном и коммутативном режимах. Сообщения включали такие команды, как «добавить контакт», «обновить контакт» и «удалить контакт». Контакты были уникально идентифицированы по GUID, что делало операции добавления тривиально идемпотентными и взаимно коммутативными.

Операции удаления требовали немного больше работы, чтобы взаимодействовать с добавлениями. Если удаление (delete) обрабатывается первым, а добавление (add) – вторым, результат должен быть таким же, как если бы они обрабатывались в обычном порядке. То есть удаление, за которым следует добавление, должно привести к тому, что контакт будет отсутствовать. Мы добились этого, сохранив список всех GUID-контактов, которые были удалены, даже если сам контакт в это время отсутствовал. А затем, когда добавление обрабатывалось, если его GUID был в этом списке (который обычно называют надгробием, показан на рис. 4.14), контакт не добавлялся.


**Рис. 4.14.** Надгробия

Сложнее всего было разобраться с операциями обновления. Как и в случае с операцией PUT в HTTP, тривиальная реализация обновления является идемпотентной, но не коммутативной. Чтобы решить эту проблему, мы присвоили каждому сообщению обновлений GUID. Каждый узел отслеживал GUID последнего обновления, которое устанавливало свойства контакта. Он также хранил список обновлений GUID, которые видел в прошлом. Когда пользователь изменял контакт, узел добавлял текущий GUID в список прошлых GUID, а затем генерировал новый текущий GUID. Он включал и текущий GUID, и список прошлых GUID в сообщение об обновлении.

Когда узел получал обновление, он сначала проверял, есть ли текущий GUID обновления в его собственном списке прошлых обновлений. Если да, то он игнорировал обновление. Если нет, он выполнял обратную проверку – был ли его текущий GUID в списке прошлых обновлений сообщения. Если вторая проверка пройдена, он принимал обновление, забирая весь список прошлых обновлений.

Пока одна из этих двух проверок проходит, обновления обмениваются друг с другом. При получении не по порядку будущее обновление будет передавать GUID прошлого обновления. Прошлое обновление впоследствии будет проигнорировано.

Однако если обе проверки не пройдут, то придется потрудиться еще. В этом сценарии обнаруживается параллельное обновление. Пользователь изменил один и тот же контакт на двух разных узлах, пока они были отключены. Наша реакция на это заключалась в объединении двух наборов свойств. Если поле, например но-

мер телефона, было одинаковым, мы сохраняли это значение. Там, где они были разные, мы просто объединяли их. Это означало, что каждое из полей допускает несколько значений. К счастью, уже понятно, что у контакта может быть несколько телефонных номеров. На рис. 4.15 показаны примеры этих трех сценариев.


**Рис. 4.15.** Сравнение прошлых и текущих GUID

В дополнение к объединению свойств узел также объединяет GUID. Список прошлых GUID был объединением текущего и входящего списков. А текущий GUID? Здесь наша структура данных была немного сложнее, чем в моем первом описании. Текущий GUID также был списком. Обычно он содержал только один элемент. Но после слияния он содержал их два (или даже больше, если были обнаружены дополнительные параллельные обновления).

Это слияние является коммутативным (игнорируя порядок конкатенации, что мы с удовольствием сделаем). Каждая сторона параллельного обновления выполняла слияние, когда видела сообщение другой стороны. Они обе получат один и тот же список прошлых GUID и один и тот же список текущих GUID. Когда пользователь впоследствии редактировал этот объединенный контакт, оба текущих GUID были бы добавлены в список прошлых. И таким образом, обе стороны с радостью заменяли свое автоматическое слияние на ручное слияние пользователя.

## Воспроизведение истории

Это решение работало довольно хорошо. В конечном итоге оно имело сильную конечную согласованность (хотя в то время мы не знали этого термина). Мы с гордостью показывали потенциальным покупателям, что можем отсоединить PDA, внести изменения, а затем синхронизировать его обратно. После обработки всех сообщений у всех клиентов был один и тот же список контактов в одном и том же состоянии.

Однако в процессе синхронизации все выглядело несколько нечетко. Если на одной стороне происходило большое количество изменений, все эти правки воспроизводились перед вашими глазами на другой стороне. Учитывая ско-

рость сетей в то время, вы могли легко прочитать список имен, которые были добавлены, изменены и впоследствии удалены, пока история воспроизводилась на устройстве.

Добавление нового устройства к этой системе выявило весь масштаб проблемы. Поскольку система была полностью основана на обработке сообщений в точности в том виде, в котором они были отправлены изначально, вся история сообщений сохранялась в центральном хранилище. Мы назвали это конвейером транзакций. Когда новое устройство вводилось в конвейер транзакций впервые, как показано на рис. 4.16, оно должно было получить и обработать каждое из этих сообщений. Это означает, что оно увидит все прошлые правки. Оно бы увидело контакты, которые уже давно были удалены. По мере роста истории время, необходимое для добавления нового устройства, возрастало пропорционально.

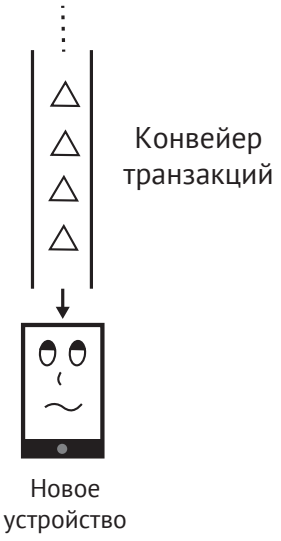


**Рис. 4.16.** В конвейер транзакций вводится новое устройство

Мой друг и я никогда не продавали эту систему управления контактами. В конце концов, она оказалась такой же неуклюжей, как и продукт Microsoft, с которым мы конкурировали. Возможно, мы могли бы найти способ обрезать историю или загружать снимки. Но если бы мы знали о бесконфликтных реплицируемых типах данных, то предложили бы лучшее решение.

## Бесконфликтные реплицированные типы данных (CRDT)

Мы достигли полезного результата для распределенной системы, основанной на обработке и пересылке сообщений. Если каждый узел видит каждое сообщение и узлы пересылают сообщения без изменений, то для достижения сильной конечной согласованности (SEC) достаточно двух свойств:

- идемпотентности (игнорирование дубликатов);
- коммутативности (независимость от порядка).

Докажите эти свойства для способа обработки сообщений узлами, и у вас уже есть очень надежная система. Я построил множество систем, используя именно эту технику. Она хорошо сочетается с такими компонентами инфраструктуры, как Amazon SQS, Rabbit MQ и MSMQ, которые обеспечивают передачу и доставку сообщений. Она требует лишь минимального набора гарантий от этих компонентов, что помогает им работать в масштабе, не становясь чрезмерно ограниченными.

Но это не самый общий результат. Мы можем оптимизировать нашу распределенную систему еще больше, если разрешим узлам изменять сообщения. Вместо того чтобы требовать, чтобы узел пересылал точно те же самые сообщения, которые он получил, мы можем позволить ему обобщить свои знания и отправлять меньше сообщений. Такая стратегия используется в бесконфликтных реплицированных типах данных (CRDT).

## CRDT, основанные на состоянии

Шапиро, Прегиса и их коллеги описали два общих решения проблемы сильной конечной согласованности – CRDT на основе состояний и CRDT на основе операций. CRDT, основанные на операциях, требуют протокола доставки, который обеспечивает доставку «один и только один раз» и сохраняет причинно-следственный порядок. Мы бы предпочли найти решение, которое не накладывает столь серьезных ограничений на компоненты инфраструктуры. К счастью, CRDT, основанные на состоянии, не имеют таких ограничений. CRDT на основе состояний и CRDT на основе операций могут эмулировать друг друга, и поэтому они эквивалентны. По этим причинам мы можем отложить CRDT на основе операций и полностью сосредоточиться на разновидностях, основанных на состояниях.

Бесконфликтный реплицированный тип данных – это структура данных, которая существует не в одном месте, как типичный абстрактный тип данных, она существует в нескольких местах. Каждый узел в распределенной системе имеет свою собственную реплику CRDT. Работа с репликой CRDT полностью соответствует нашим требованиям к независимости от местоположения. Каждая реплика может обслуживать запросы изолированно, не связываясь с другими репликами. И каждая реплика может обрабатывать команды, которые немедленно изменяют ее состояние. Эффекты этих команд будут переданы другим репликам в конечном итоге согласованным образом. Все реплики придут к одному и тому же состоянию, когда все обновления будут доставлены.

Главное – понять, что значит «обновление доставлено».

### Частично упорядоченное состояние

Каждая реплика CRDT, основанная на состоянии, имеет внутреннее состояние. Как разработчик приложения вы выбираете форму этого внутреннего состояния. Она основывается на проблеме, которую вы пытаетесь решить. Но это состояние должно удовлетворять нескольким условиям:

- оно должно поддерживать отношение «произошло раньше» (причинность), которое определяет частичный порядок;
- все обновления должны увеличивать состояние в этом частичном порядке (предыдущая версия «произошла раньше»);

- оно должно поддерживать операцию слияния, которая берет два состояния и производит новое, которое больше их обоих (обе предыдущие версии «произошли раньше» объединенной версии).

Чтобы быть полезным, отношение «произошло раньше» должно помочь нам обнаружить параллельные обновления. Мы хотим избежать создания полного порядка и вместо этого зафиксировать частичный порядок, присущий причинности.

В отличие от нашей распределенной системы на основе ретрансляции, обновления не обязательно должны быть идемпотентными или коммутативными. Это связано с тем, что обновления будут выполняться только на одной реплике. В рамках одного процесса мы можем легко контролировать, сколько раз и в каком порядке будут выполняться обновления. CRDT не полагается на передачу сообщений, как система, которую мы только что проанализировали.

Шапиро, Прегиса и их коллеги доказали, что этих трех условий достаточно, чтобы гарантировать SEC. Все реплики сходятся к одному и тому же состоянию после доставки всех обновлений. Итак, что означает, что обновление доставлено в реплику?

## Причинная история

Когда мы исследуем состояние реплики в рамках одного процесса, мы обнаружим только две операции, вызывающие ее изменение, – обновление и слияние. Обновление происходит, когда узел выполняет некоторую команду извне сети. Возможно, на узле запущено клиентское приложение, и оно отвечает на ввод данных пользователем. Слияние происходит, когда другой узел делится состоянием своей реплики. Это происходит при *приеме* (Receive) операции сетевой коммуникации.

Вспоминая определение причинности Лесли Лэмпорта (Leslie Lamport), мы можем сказать, что состояние реплики после обновления вызвано обновлением, обновление находится в его причинной истории. Лэмпорт также показал, что операция отправки (Send) сетевого сообщения вызывает его прием (Receive). Таким образом, обновления, произошедшие на исходном узле перед отправкой, показанные на рис. 4.17, находятся в причинной истории объединенного состояния.

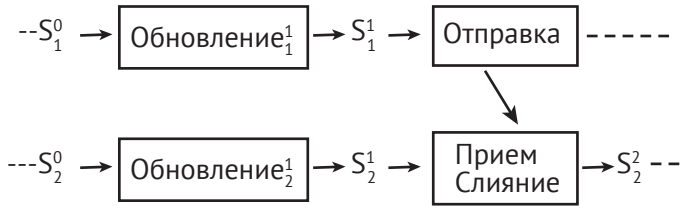


**Рис. 4.17.** Оба обновления находятся в причинной истории s22

Следуя этой логике, причинная история реплики включает в себя:

1) все обновления, которые ранее произошли в этом узле;
2) все обновления в причинных историях состояний, которые были объединены из других узлов.

Помните, что сами обновления не передаются между узлами. Только результирующие состояния. Необходимые условия, налагаемые частичным упорядочиванием состояний, гарантируют, что узлы имеют достаточно информации об обновлениях, которые были выполнены на них.

И вот теперь мы наконец-то можем ответить на вопрос. Что значит «обновление доставлено в реплику»? Это означает, что обновление находится в причинной истории ее текущего состояния.

## Векторные часы

Многовато для обработки. Давайте сделаем это более конкретно. Как мы могли бы реализовать систему управления контактами в виде CRDT? Для простоты рассмотрим только случай обновления контакта, которое изменяет свойства контакта. Мы будем представлять каждый контакт как CRDT, где каждая рабочая станция, PDA и серверный узел имеют свою собственную реплику.

Начнем с определения состояния. Каждый контакт будет иметь набор свойств, таких как имя, номер телефона и адрес электронной почты. Мы будем хранить их в состоянии.

Согласно нашему первому условию, состояние должно поддерживать оператор «произошло раньше», чтобы обеспечить нам причинность. Очевидно, просто глядя на свойства контакта, мы не можем сказать, какая из двух версий была первой. Для этого нам нужно хранить какой-то номер версии. Монотонно возрастающий номер версии будет удовлетворять нашему первому и второму условиям. Мы сможем увидеть, какая версия была первой, и будем увеличивать номер версии с каждым обновлением.

К сожалению, простой номер версии не помогает нам определить параллельные обновления. Он не отражает истинный порядок причинно-следственных связей. Вместо этого мы будем хранить отдельный номер версии для каждого узла. Это структура данных, известная как *векторные часы*[1]. На рис. 4.18 показан пример.

```
{
Узел 1:3
Узел 2:1
Узел 3:7
}

Имя: "Роберт"
Телефон: "555-1212"
Email: "rob@email.com"
```

**Рис. 4.18.** Векторные часы как часть контакта CRDT

Для сравнения двух векторных часов мы сравниваем номер версии каждого узла. Если все номера версий в первых векторных часах меньше или рав-

[1] *Friedmann Mattern.* Virtual time and global states of distributed systems. In Int. W. on Parallel and Distributed Algorithms, p. 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

ны вторым, тогда первые часы «произошли раньше» вторых. Это частичная упорядоченность, потому что возможно, что версия одного узла будет меньше в одних, а версия другого узла меньше в других. Когда это происходит, эти двое часов не имеют причинно-следственной связи.

Когда узел обновляет контакт, он увеличивает свой собственный номер версии в векторных часах, как показано на рис. 4.19. Это гарантирует, что новая версия имеет больший номер, как того требует второе условие. При каждом обновлении состояние перемещается вперед в причинном времени.

```
{                              {
Узел 1:3                       Узел 1:3
Узел 2:1                       Узел 2:2
Узел 3:7                       Узел 3:7
}                              }

Имя: «Роберт»                  Имя: «Роберт»
Телефон: «555-1212»            Телефон: «555-1212»
Email: « rob@email.com»        Email: « rob@email.com»
```

**Рис. 4.19.** Узел 2 обновляет электронную почту контакта и увеличивает свой собственный номер версии

Когда узел объединяет свое состояние с другим узлом, как на рис. 4.20, он берет максимальное значение каждого из номеров версий. Это гарантирует, что новые векторные часы больше, чем каждые из двух исходных векторных часов. Строго говоря, если один вектор уже «произошел раньше» другого, то слияние просто даст нам больший из них. В этом случае он не больше, а равен более поздней версии. На самом деле это вычисление наименьшей верхней границы двух векторов. Это более точно, чем требуется условиями SEC.

Мы видим, что векторные часы удовлетворяют необходимым условиям для SEC. Поскольку мы поместили векторные часы в состояние нашего CRDT, у нас есть способ сравнить две версии, чтобы узнать, какая из них была раньше. Увеличение версии при обновлении производит векторные часы, которые больше, чем предыдущие, поэтому обновления увеличивают состояние. Слияние двух векторных часов дает набор чисел, которые все больше или равны двум предыдущим векторным часам, и поэтому слияние производит состояние, большее, чем исходные состояния.

Самое главное – векторные часы фиксируют причинную историю реплики. Во время слияния мы можем обнаружить параллельные обновления. Если все номера версий в векторных часах для одного из двух состояний больше или равны другому, то это состояние представляет собой более свежую версию. Значения свойств контакта будут просто скопированы из большего из двух состояний. Но если ни одни из векторных часов не «произошли раньше» других, то произошло параллельное обновление. Это говорит о том, что нам нужно объединить свойства контакта.

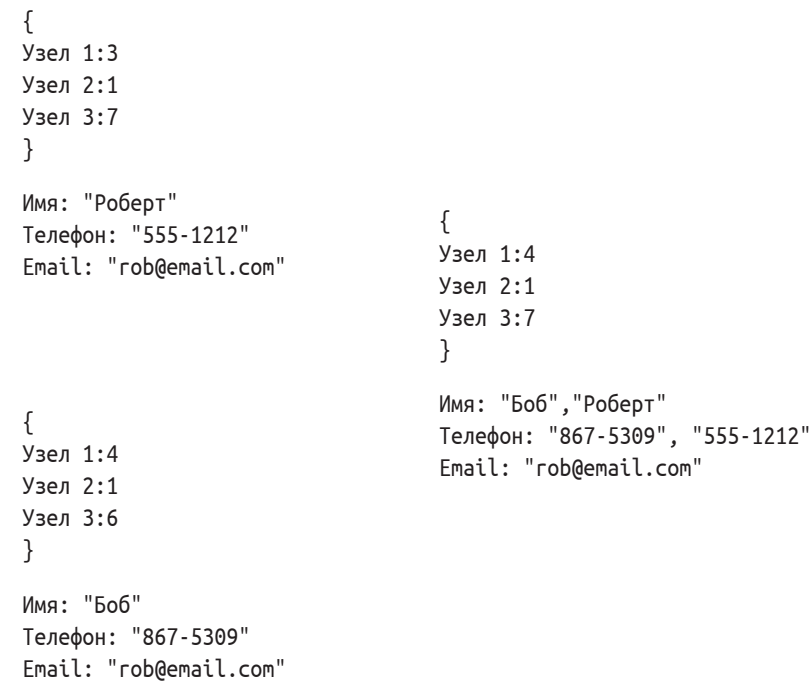


**Рис. 4.20.** При объединении двух контактов берется максимум из всех номеров версий

Векторные часы – это инструмент для построения CRDT на основе состояний. Они дают нам возможность определить частичную упорядоченность между состояниями, которая поддерживает операции обновления и слияния. При правильном использовании реплики, включающие векторные часы, будут сходиться к одному и тому же значению, когда все обновления появляются в их причинной истории.

Если бы мы с моим другом построили системы управления контактами с использованием векторных часов, то внедрение нового устройства в систему стало бы простой загрузкой. Устройство получало бы только текущее состояние каждого контакта и векторные часы, представляющие причинную историю этого состояния. Когда пользователь внесет изменения в это новое устройство, оно просто добавит себя к векторным часам в версии 1 и поделится своим новым значением.

## История фактов

Векторные часы – не единственный бесконфликтный реплицируемый тип данных. Любая структура данных, которая частично упорядочена, увеличивается при обновлении и вычисляет наименьшую верхнюю границу при слиянии, может быть использована в качестве CRDT. Исследователи уже определили

множество таких структур данных для использования в различных ситуациях[1]. Немного поработав, вы сможете разработать CRDT, который точно подходит именно для вас.

Есть одна структура данных, которая применима в удивительно большом количестве случаев. Она имеет четко определенную частичную упорядоченность. Она увеличивается при обновлении. И она обладает допустимой операцией слияния.

Я говорю, конечно, о скромном наборе.

## Наборы

Набор – это коллекция предметов, которая обладает несколькими интересными свойствами:

1) она не содержит дубликатов;
2) она не упорядочена.

Первое свойство означает, что элемент либо есть в наборе, либо его нет. Набор не запоминает, сколько раз мы пытались добавить элемент. Второе свойство говорит нам, что ни один элемент не идет ни до, ни после другого элемента. Набор не помнит, в каком порядке мы добавляли элементы.

Интересно заметить, что вставка набора удовлетворяет двум условиям, необходимым для SEC в системе ретрансляции. Благодаря первому свойству вставка набора является идемпотентной. Если мы попытаемся вставить элемент, который уже есть в наборе, он останется неизменным. А из-за второго свойства вставка набора является коммутативной. Вставка элементов в противоположном порядке дает тот же самый набор. Эти два свойства означают, что вставка набора ведет себя хорошо при наличии повторяющихся или неупорядоченных сообщений.

Идемпотентность и коммутативность не требуются для использования набора в качестве CRDT, основанного на операциях. Свойства, которые требуются для CRDT, – это частичная упорядоченность, возрастание обновления и слияние, которое вычисляет наименьшую верхнюю границу. До тех пор, пока мы не разрешаем удалять элементы из набора, мы можем легко достичь всех трех свойств.

### Частичная упорядоченность

Наборы частично упорядочены по отношению подмножества. Одно множество является подмножеством другого, если оно содержит только те элементы, которые могут быть найдены в другом. Если мы используем подмножество как «произошло раньше», то мы определили частичную упорядоченность.

Возьмем, например, набор {🚗, 🚲, 🚜}. Он является подмножеством {🚗, 🚲, 🚂, 🚜}. Каждый элемент первого из них также находится во втором.

Теперь рассмотрим третий набор {🚲, 🚂, 🚜}. Он также является подмножеством второго. Но ни первый, ни третий не является подмножеством другого.

[1] *Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski.* A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, p. 50. ffinria-00555588f.

Первый набор содержит элемент, которого нет в третьем (🚗), а третий содержит элемент, которого нет в первом (🚂). Следовательно, они не связаны отношением подмножества.

Тот факт, что некоторые наборы могут быть сопоставлены, а другие нет, делает их частично упорядоченными. Мы можем использовать этот частичный порядок для представления причинности. Поднабор «произошел раньше» набора.

### Обновление

Единственная операция обновления, которую мы разрешаем для набора, – это вставка. Если вы попытаетесь вставить элемент, который уже содержится в наборе, то набор не изменится. Но если элемент еще не был в наборе, то новый набор будет содержать все, что было в старом наборе, плюс один дополнительный элемент. Поэтому вставка набора, когда она изменяет набор, увеличивает его значение в рамках частичной упорядоченности.

Если вы думаете о наборе как о содержащем все знания реплики, вы можете увидеть, что вставка набора увеличивает эти знания. Реплика попрежнему знает все, что она знала раньше. Но после обновления она знает немного больше.

Это также иллюстрирует, как набор может представлять причинную историю реплики. Вспомним, что причинная история реплики включает все обновления, которые произошли в ее прошлом. Перечисляя элементы набора, вы можете четко увидеть все вставки, которые произошли с течением времени.

### Слияние

Допустимая операция слияния в CRDT вычисляет наименьшую верхнюю границу двух значений. Наименьшей верхней границей двух множеств при частичной упорядоченности подмножеств будет то множество, которое содержит все элементы обоих множеств. Это будет супермножество каждого из них. Чтобы вычислить наименьшую верхнюю границу, мы просто должны взять объединение множеств.

Рассмотрим снова два множества {🚗, 🚲, 🚜} и {🚲, 🚂, 🚜}. Ни одно из них не является подмножеством другого. Но мы можем вычислить наименьшее множество, которое является супермножеством обоих. Это будет объединение множеств – {🚗, 🚲, 🚂, 🚜}.

Это соответствует нашей интуиции о слиянии, а также условиям, необходимым для SEC. Если один узел объединяет все знания удаленного узла в свои собственные, то в итоге получается объединение двух знаний. В результате этого слияния знания увеличиваются.

Представление об этом как о комбинации двух причинно-следственных историй также имеет интуитивный смысл. Причинная история после слияния включает все обновления, которые произошли как локально, так и удаленно.

## Исторические записи

Давайте формализуем эту интуицию о множестве, представляющем причинную историю реплики. Вместо того чтобы рассматривать наборы транс-

портных эмодзи, пусть элементами набора будут реальные исторические записи.

Когда пользователь выполняет действие на узле, мы фиксируем это действие как историческую запись. Мы записываем, что он пытался сделать и какие опции или параметры он выбрал при выполнении этого действия. Запись – это просто структура, которая собирает всю эту информацию. Она фиксирует все необходимые данные, которые были частью решения пользователя.

Когда мы соединим эти исторические записи в единое целое, чтобы сформировать причинную историю, мы заметим четыре вещи:

1) мы должны различать похожие записи;
2) мы не можем удалить запись, если она вставлена в набор;
3) мы не можем изменить запись, которая уже является частью набора;
4) некоторые записи связаны друг с другом.

Эти четыре наблюдения дают начало правилам исторических моделей.

## Различение записей

Обращаясь к системе управления контактами, определим первое действие, которое пользователь может выполнить на узле. Он захочет создать новый контакт. Когда новый контакт создается, он не имеет никаких свойств. Они могут быть добавлены позже.

Причинная история, которую создает пользователь, будет выглядеть следующим образом:

```
{ContactCreation}
```

Однако когда пользователь пытается создать второй контакт, он сталкивается с проблемой. Запись `{ContactCreation}` уже существует в наборе. Ее нельзя вставить снова.

Чтобы вставить больше записей о создании контактов в причинную историю, нам нужно различать их. Здесь система, зависящая от местоположения, использовала бы автоинкрементный идентификатор. Это позволило бы создать причинную историю, которая выглядит так:

```
{ContactCreation(1),ContactCreation(2)}
```

Проблема с этой стратегией становится очевидной, когда мы объединяем причинную историю одного узла с причинной историей другого. Слияние выполняется с помощью объединения множеств. Если бы они оба генерировали идентификаторы, используя автоинкрементный счетчик, то выдавали бы один и тот же идентификатор для разных контактов. Объединение множеств объединило бы разные контакты в один.

Вместо этого узел выбирает идентификатор, не зависящий от местоположения. Естественный ключ был бы лучшим вариантом, но в данном случае у нас в проблемной области нет неизменяемого естественного идентификатора. Мы не спрашиваем у контактов дату рождения, государственный идентификационный номер и образец ДНК. Нам придется довольствоваться GUID.

```
{ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb),
ContactCreation(5853e3fe-059a-4180-af0a-f969260be882)}
```

Итак, мы выяснили, что историческая запись уникально идентифицируется только по ее содержанию. Других идентификаторов у нее нет. Это делается для того, чтобы гарантировать, что причинные истории объединяются независимым от местоположения образом.

## Удаление записи

Следующим действием, которое хочет выполнить пользователь системы управления контактами, является удаление контакта. Наиболее естественный способ представления удаления – удаление создания контакта из причинной истории. Удаление второго контакта (5853…) из набора приведет нас к следующему:

```
{ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb)}
```

Но эта стратегия не сработает. Она нарушает второе условие CRDT, основанного на состоянии. Операции обновления могут только увеличивать состояние реплики в пределах частичного порядка. Удаление элемента из набора создает подмножество, а не супермножество. Это переносит состояние назад в причинном времени.

Становится очевидным, что мы совершили ошибку, когда мы делимся состоянием с другими узлами. Предположим, пользователь создает контакт, а затем его устройство делится своим состоянием с другим узлом. Это создание контакта теперь является частью реплики другого узла.

Теперь предположим, что пользователь удаляет этот контакт, и узел неверно представляет это, удалив его из набора. Когда удаленный узел в какой-то момент в будущем поделится своим состоянием с узлом пользователя, наборы реплик будут объединены. Контакт, который был удален, внезапно появится вновь. Просто наберите в поисковике «удаленный контакт появляется снова», чтобы увидеть, насколько распространена эта ошибка.

Вместо того чтобы удалять историческую запись из причинной истории, мы должны создать новую историческую запись. Эта новая запись представляет собой удаление контакта.

```
{ContactCreation(74aac247-a86a-4af8-9db4-cf1387f8a1fb),
ContactCreation(5853e3fe-059a-4180-af0a-f969260be882),
ContactDeletion(5853e3fe-059a-4180-af0a-f969260be882)}
```

Мы восстановили условие, при котором обновления увеличивают состояние внутри частичной упорядоченности. Этот новый набор является супермножеством предыдущего. И слияние состояния с другими узлами никогда не приведет к повторному появлению контакта.

Мы только что обнаружили, что историческая запись не может быть удалена. Запись может *представлять собой* удаление сущности. Но она не может быть удалена из причинной истории.

### Изменение записи

Следующим действием, которое пользователь может захотеть выполнить на узле, будет установка свойств контакта. Когда он это сделает, мы сохраним историческую запись его действий. Она включает, какой контакт они изменяют и какие значения они установили для этих свойств. Результирующая причинная история выглядит примерно так:

```
{ContactCreation(74aa...),
ContactModification(74aa...,»Bob»,»555-1212»)}
```

При втором редактировании наивным решением будет изменение записи в наборе:

```
{ContactCreation(74aa...),
ContactModification(74aa...,»Robert»,»555-1212»)}
```

Мы уже видим, почему это не работает. Новый набор не является супермножеством исходного. Это нарушает второе условие – обновления должны увеличивать состояние частичной упорядоченности. Фактически мы создали новое множество, которое не связано причинно со старым.

Чтобы решить эту проблему, мы можем частично упорядочить записи модификаций. Один из способов сделать это – добавить векторные часы:

```
{ContactCreation(74aa...),
ContactModification(74aa...,[node1:1],»Bob»,»555-1212»),
ContactModification(74aa...,[node1:2],»Robert»,»555-1212»)}
```

Мы обнаружили, что если запись стала частью истории, то ее нельзя изменить. Мы можем добавить новую запись, которая *представляет собой* модификацию сущности, но старые записи должны остаться.

Поскольку у нас уже есть полный набор исторических записей, векторные часы являются излишними. Помните, что векторные часы помогают нам превратить простую структуру данных в CRDT. Они фиксируют частичную упорядоченность причинности. Но теперь, когда мы работаем с набором простых структур данных, мы можем полагаться на этот набор для фиксации причинности. У нас есть возможность небольшой оптимизации.

### Записи причинно-следственно связаны

При внимательном рассмотрении набора исторических записей можно выявить несколько отношений. Наиболее очевидное из них заключается в том, что записи изменений контакта `ContactModification` содержат GUID записи создания контакта `ContactCreation`. Как показано на рис. 4.21, мы можем представить эту взаимосвязь напрямую, нарисовав стрелку от изменения к созданию.

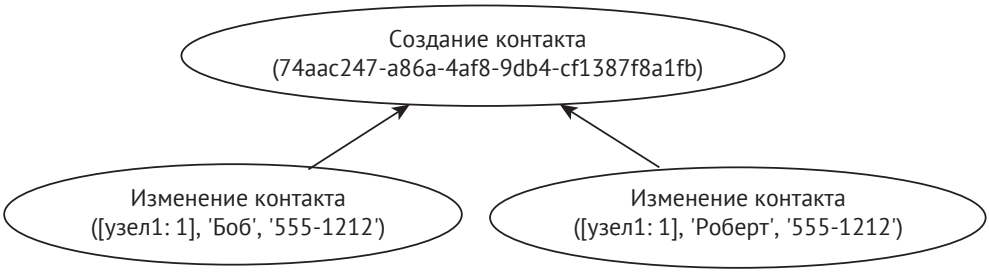


**Рис. 4.21.** Создание контакта предшествует его изменению

Этот граф по-прежнему является множеством. Каждая вершина графа – это историческая запись в множестве. Все, что мы сделали, – это заменили подразумеваемые отношения общих GUID стрелками.

Второе наблюдение, которое мы можем сделать, заключается в том, что векторные часы на самом деле являются ссылками друг на друга. Часы [node1: 2] представляют обновление, произошедшее на узле 1, увеличивающее его версию с 1 до 2. Они ссылаются на предыдущие часы [node1: 1] путем умозаключения. Как показано на рис. 4.22, мы можем заменить векторные часы стрелками.


**Рис. 4.22.** Векторные часы заменены стрелками

Замена векторных часов стрелками сохраняет частичную упорядоченность между изменениями. Нам по-прежнему легко сравнить две записи `ContactModification`, чтобы увидеть, какая из них появилась раньше другой. Если мы можем проследить путь от одной к другой вдоль стрелки *в правильном направлении*, то запись в начале последней стрелки «произошла раньше» записи в хвосте первой стрелки.

Граф отражает частичную упорядоченность причинности.

Это верно в целом, а не только для изменений. Создание контакта (`ContactCreation`) произошло до его изменения (`ContactModification`). Пользователь должен был создать контакт, прежде чем установить его свойства. Если контакт не был создан на локальном узле, то он должен быть создан удаленно и объединен в набор. Согласно причинности Лампорта, даже это удаленное создание произошло до локального изменения.

### Преимущества явной причинности

Мы изобразили причинно-следственные связи между историческими записями в виде стрелок в направленном графе. Это не просто оптимизация. Это также обеспечивает соблюдение предварительных условий действий пользователя. Пользователь не может изменять свойства контакта, который еще не был создан. Он также не может удалить несуществующий контакт. Когда причинно-следственные связи между записями подразумевались только общим идентификатором GUID, структура данных ничего не делала, чтобы помочь нам обеспечить выполнение этих предварительных условий. Но теперь, когда она явно фиксирует стрелки, эти предварительные условия обеспечиваются существованием записи в головной части стрелки.

Еще одно преимущество заключается в том, что мы только что заменили менее важную информацию более важной. Векторные часы включали в себя имена узлов, в которых были произведены изменения. Эта информация была нужна только для того, чтобы узел знал, какой номер версии увеличивать при обновлении. После этого имена узлов стали не важны. Стрелки отбрасывают имена узлов в пользу явных ссылок на предыдущие версии. Не имеет значения, была ли предыдущая версия создана на том же самом узле или появилась в результате слияния.

Взамен отброшенной неважной информации стрелки предоставляют нам гораздо более важную информацию. Они говорят о том, как изменилась сущность с течением времени. Это полезно для вычисления лучшего слияния.

Предположим, что после создания и инициализации контакта он делится информацией с удаленным узлом. В этой реплике другой пользователь вносит другое изменение. После того как локальный пользователь получает слияние, он видит граф, показанный на рис. 4.23.

Частичная упорядоченность изменений показывает нам, что два листа графа не связаны причинно-следственно. Ни один из них не произошел раньше другого. Поэтому нам необходимо объединить два набора свойств для отображения пользователю.

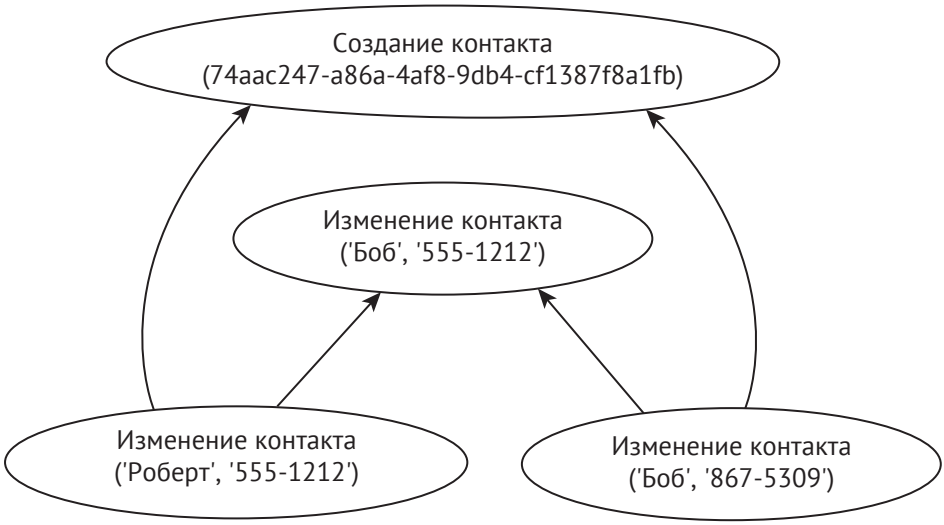


**Рис. 4.23.** Граф после слияния параллельных изменений

Слияние на основе конкатенации, которое мы делали раньше, создаст контакт с двумя именами и двумя номерами телефонов. Если все, что у нас было, – это две структуры данных и двое векторных часов, это лучшее, чего мы могли бы надеяться достичь. Но наличие графа дает третью точку данных. Мы можем увидеть *ближайшего общего предка* двух листьев.

Сравнивая левую ветвь с ближайшим общим предком, мы видим, что локальный пользователь изменил только имя. А сравнение правой ветви показывает, что удаленный пользователь изменил лишь номер телефона. Это позволяет нам выполнить гораздо более разумное слияние – отображение самого последнего имени и самого последнего номера телефона:

```
имя: «Роберт»;
телефон: «867-5309».
```

Это трехстороннее слияние происходит только на дисплее. Набор (или граф) исторических записей никак не изменяется. Более того, все узлы выполняют это трехстороннее слияние совершенно одинаково. Все они имеют один и тот же граф, поэтому все они вычисляют один и тот же результат. Таким образом, даже когда история захватывает причинно несвязанные записи, это не приводит к конфликту. Каждый узел сходится к одному и тому же значению.

С помощью этого последнего наблюдения мы обнаружили, что исторические записи связаны друг с другом в явном виде. Эта связь отражает частичную упорядоченность причинности. Запись в начале стрелки произошла раньше той, что в хвосте.

## Исторические факты

Поскольку эти записи больше не являются простыми плоскими структурами данных, я не решаюсь называть их записями. Мне также не нравится называть их историческими *событиями*, так как это вызывает в памяти поиск событий. Поиск событий фиксирует полностью упорядоченную последовательность исторических событий, но не фиксирует явным образом отношения *между* историческими событиями. Поэтому я называю эти элементы причинного набора *историческими фактами*.

Стрелки указывают на *предшественника* исторического факта. Предыдущий факт предшествовал более позднему. Последний я называю *преемником*. Стрелки вставляются в граф только вместе с преемником и никогда после него. Это обеспечивает соблюдение предварительных условий, сохраняет частичную упорядоченность причинно-следственных отношений, а также предотвращает циклы. Содержание факта в сочетании с его набором стрелок-предшественников – это все, что отличает его от других фактов. Поскольку факты не имеют внешней идентичности, узлы могут ссылаться на факты независимо от местоположения. Весь набор причинно-следственных связей – это то, что я называю *исторической моделью*.

**Историческая модель** – это CRDT, основанный на состоянии, который отражает причинно-следственные связи между историческими фактами в виде направленного ациклического графа. Стрелки графа вводят частичную упорядоченность, которая показывает, какие факты произошли раньше, чем другие факты. На факты в графе можно ссылаться, запрашивать и использовать их без зависимости от местоположения узла.

Граф поддерживает две операции – вставку и слияние. Вставка нового исторического факта перемещает граф вперед по причинно-следственной временной шкале, поскольку полученный граф является супермножеством исходного. Слияние двух исторических моделей вычисляет их наименьшую верхнюю границу и, следовательно, помогает удаленным репликам достичь сильной конечной согласованности.

## Заключение

Мы определили свойства, которыми должна обладать распределенная система, чтобы быть независимой от местоположения. Они должны демонстрировать независимость от местоположения как в идентификации, так и в поведении.

Независимая от местоположения идентификация не подразумевает привязки объекта к его местоположению. Любой узел должен иметь возможность генерировать и сравнивать уникальный неизменяемый идентификатор для нового объекта без связи с другими узлами. Это уменьшает задержку и увеличивает автономность изолированных узлов в распределенной системе.

Независимое от местоположения поведение позволяет узлу запрашивать и совершать операции с репликами объектов в изоляции. Эти реплики достигают сильной конечной согласованности, когда все реплики приходят к одному

и тому же состоянию после доставки всех обновлений. Это может быть достигнуто с помощью идемпотентной и коммутативной системы ретрансляции или более сложного бесконфликтного реплицируемого типа данных.

Из этих ограничений мы вывели набор правил, которые помогают определить системы, работающие независимо от местоположения их узлов. Я называю этот набор правил историческим моделированием. Рассуждения, изложенные в данной главе, демонстрируют, что историческая модель удовлетворяет требованиям сильной конечной согласованности.

Но прежде, чем создавать исторические модели, мы должны понять, что именно они говорят нам о системах, которые мы собираемся построить. Давайте сначала изучим эти правила как средство анализа проблемы. Среди прочего правила быстро показывают, когда мы делаем предположение о наличии у данных местоположения. Такой анализ позволит выявить потенциальные проблемы задолго до того, как они смогут вызвать неполадки при создании систем.

# Глава 5

# Анализ

Неизменяемая архитектура гарантирует, что мы строим систему в рамках ограничений, которые компьютеры могут легко преодолеть. Когда мы начинаем с предположения о неизменяемости, мы уверены в том, что результирующая система будет обладать желаемыми характеристиками. Теперь остается понять, как анализировать проблемную область в рамках этих ограничений. Мы должны обнаружить элементы проблемы и выразить их в виде неизменяемых единиц.

Основная цель анализа – обеспечить общее понимание между командой разработчиков и владельцем продукта. Аналитик выделяет суть проблемной области и документирует это понимание таким образом, чтобы владелец продукта мог его проверить. Чтобы это было эффективно, анализ должен быть понятен без обучения или жаргона.

Вторая цель – зафиксировать ожидаемое поведение, чтобы его можно было правильно реализовать и протестировать. Для этого анализ должен быть задокументирован и представлен разработчикам и тестировщикам. Результаты анализа должны быть достаточно точными, чтобы неправильная реализация привела к провалу этих тестов. Они должны направлять и ограничивать разработку и отвечать на вопросы, возникающие в ходе реализации.

Поэтому средства связи должны удовлетворять несколько аудиторий одновременно. Результаты анализа должны быть легко понятными экспертам в любой области бизнеса, а не только в технических областях. Они также должны быть достаточно точными, чтобы генерировать значимые тесты. И они должны представлять идеи, которые действительно могут быть реализованы. Сейчас наша задача состоит в том, чтобы определить язык, который решит все эти проблемы.

Требования и документы анализа обычно пишутся в прозе. К сожалению, естественные человеческие языки – плохой выбор, когда целями являются точность и однозначность коммуникации. Проза содержит предположения, которые автор не может сформулировать, а читатели не могут распознать. Она содержит жаргон, который передает смысл для одних, но запутывает других. А человеческие языки не компилируются, они не ограничиваются описанием только тех решений, которые могут быть реализованы, и тех, которые не могут быть выполнены для проверки данного решения.

Поэтому артефакты нашего проекта не будут написаны на английском или любом другом естественном языке. Они будут написаны на точном языке ма-

тематики. И все же, чтобы избавить артефакты от жаргона, требующего математического образования, они будут ограничены очень простыми понятиями. Язык анализа точно соответствует языку исторического моделирования. Он производит набор утверждений, которые могут быть однозначно проверены. Он также может описывать только те системы, которые могут быть реализованы в рамках ограничений распределенных систем.

## Примеры использования

В 1992 году Ивар Якобсон (Ivar Jacobson) опубликовал первое руководство по «анализу примеров использования». *Объектно-ориентированная разработка программного обеспечения* (Object-Oriented Software Engineering – OOSE)[1] описывает систему для документирования поведения программного обеспечения. Она разбивает это поведение на целенаправленные единицы, называемые «сценариями использования». В каждом примере использования субъект использует систему для достижения цели. Хотя первоначально Якобсон разрабатывал сценарии использования как часть процесса под названием Objectory (Фабрика объектов), позже он объединил усилия с Грейди Бучем (Grady Booch) и Джеймсом Румбо (James Rumbaugh), чтобы привнести моделирование и анализ вариантов использования в унифицированный язык моделирования (Unified Modeling Language – UML).

Хотя большая часть OOSE сосредоточена на подробном текстовом языке для описания вариантов использования, в книге также представлен высокоуровневый графический язык, называемый диаграммами вариантов использования. Цель диаграммы вариантов использования состоит в том, чтобы проиллюстрировать отношения между несколькими вариантами использования и людьми, которые их инициируют. Варианты использования могут быть связаны друг с другом несколькими способами. Более крупный вариант может *включать* в себя несколько более мелких вариантов, если более мелкие варианты выполняются в процессе достижения более крупного результата. Родительский вариант использования может *обобщать* более конкретные варианты, чтобы собрать их общие атрибуты и шаги. И более подробный вариант использования может *расширять* менее подробный вариант, предоставляя дополнительные шаги, которые выполняются при определенных условиях.

На практике диаграммы вариантов использования нашли гораздо меньшее применение, чем их более подробные текстовые аналоги. Во многих диаграммах требований, которые я читал, диаграмма вариантов использования включается в основном в качестве оглавления, если она вообще нарисована. Мартин Фаулер (Martin Fowler) даже выразил мнение, что «эти диаграммы не представляют особой ценности»[2], сосредоточившись вместо этого на гораздо более подробной текстовой форме сценариев использования.

Но, учитывая проблемы с описанием поведения в прозе, возможно, следует уделить больше внимания графическому языку. Если мы решим некоторые

[1] *Ivar Jacobson.* Object Oriented Software Engineering: A Use Case Driven Approach Revised Printing edition. 1992. Addison-Wesley.

[2] https://martinfowler.com/bliki/UseCase.html.

проблемы с графическим языком, то он может стать точным, однозначным и легко понимаемым артефактом. На основе оригинальной работы Якобсона мы можем определить диаграмму, которая решает ту же самую задачу, но таким образом, чтобы обеспечить более ценную основу для системного анализа. Мы можем достичь этого путем унификации размера действий и отношений между действиями.

## От сценария использования к решению

Первая проблема, которую мы можем решить, заключается в том, что случаи использования могут иметь несколько различных размеров. Чтобы решить эту проблему, мы можем изменить характер диаграммы, дабы уменьшить размер каждого действия до атомарной единицы. Традиционно сценарий использования концентрируется вокруг одной цели. Достижение этой цели требует нескольких шагов. Составление диаграмм многоэтапных сценариев использования увеличивает избыточность, несогласованность и сложность.

Избыточность возрастает, поскольку сценарии использования могут иметь общие шаги. Вариант использования, содержащий несколько шагов, может прерываться или разветвляться на несколько сценариев. Примерами прерывания может быть отказ в авторизации, низкий уровень запасов или необходимость утверждения. Если шаги прерваны, в примере использования перечисляются шаги восстановления в качестве альтернативы основному успешному потоку. Вариант сценария использования является менее серьезным, чем полное прерывание. Он запускается некоторым условием, которое требует дополнительных шагов. Если эти шаги просты, они могут быть перечислены в рамках одного сценария использования. Если они сложны, то, возможно, их лучше всего будет разбить на дополнительные варианты использования. В любом случае, сценарий использования может легко превратиться в пересекающуюся коллекцию потоков.

Несогласованность возникает из-за размера и объема примеров использования. Многоэтапные примеры использования не имеют одинакового размера. Два человека, анализирующих одну и ту же проблему, могут прийти к совершенно разной детализации примеров использования. Даже один человек, разбивающий проблему на части, может выбрать небольшие варианты использования для действий, которые он знает до мельчайших подробностей, и большие варианты использования для вещей, которые он еще не до конца понимает. Сама структура диаграммы вариантов использования показывает, что размер не всегда одинаков – одни варианты использования включают в себя другие. Меньшие шаги сценария использования появляются как часть более крупного сценария.

Сложность является следствием наличия множества взаимосвязанных шагов в рамках одного варианта использования. Команды разработчиков часто сталкиваются с тем, что сценарии использования слишком большие и сложные, чтобы оценить и завершить их за одну итерацию. По этой причине сценарии использования часто разбиваются на пользовательские истории. Пользовательская история гораздо ближе к единому атомарному решению, которое принимает пользователь и на которое реагирует система. Часто существует

необходимое состояние, в котором уже должна находиться система, чтобы пользовательская история имела смысл. Это состояние может быть достигнуто путем выполнения предыдущих пользовательских историй, которые уже были завершены во время предыдущих итераций разработки. Такой уровень детализации позволяет команде планировать, отслеживать и выполнять свою работу. Невыполненная работа по продукту будет содержать истории пользователей, но не сценарии использования.

Настаивая на том, чтобы действие было атомарной единицей, мы приближаемся к более унифицированной единице размера. Каждый пузырек представляет не весь сценарий использования, а одно атомарное решение. На рис. 5.1 слева показан один сценарий использования. Справа перечислены отдельные решения, которые пользователь принял для достижения цели, указанной в сценарии использования.

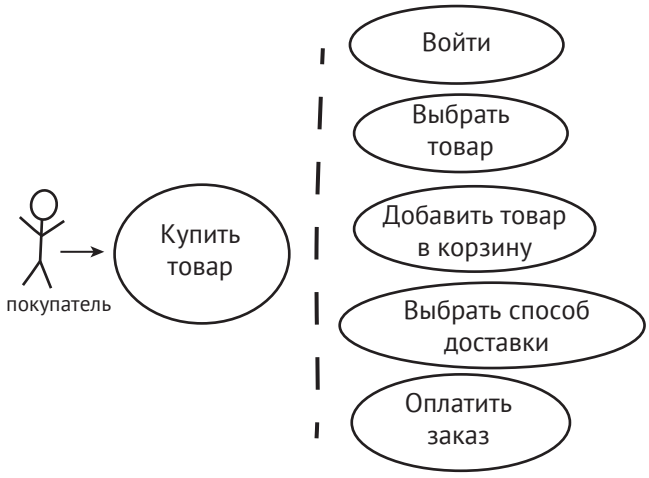


**Рис. 5.1.** Сценарий использования разбивается на несколько атомарных решений

Разбивка сценариев использования на атомарные решения повышает ценность итоговой диаграммы. Краевые случаи и сценарии восстановления могут быть более четко изображены и понятны. Каждое конечное действие имеет одинаковый размер, все они представляют собой одно атомарное решение. Команда может более четко видеть, как планировать свою работу для достижения целей, изображенных на диаграмме.

## От расширения к преемственности

Вторая проблема с диаграммой вариантов использования заключается в том, что между сценариями использования можно установить несколько различных отношений. Сценарии использования могут расширять, включать или обобщать друг друга. Понимание различий между этими тремя отношениями требует дополнительного обучения. Это вводит жаргон о процессе составления диаграмм вариантов использования, который не является важной частью моделируемой области.

Однако когда сценарии использования разбиваются до уровня атомарных решений, картина становится немного проще. Решения имеют только одно отношение – они предшествуют друг другу. Решение, которое должно быть выполнено до того, как начнется выполнение другого решения, является *предпосылкой*. Последовательность решений от предпосылки к последующему рассказывает историю о том, как субъект достигает цели.

Отношение предпосылки является строгим. Когда аналитик проводит линию от решения к его предпосылке, он утверждает, что предпосылка должна быть выполнена, прежде чем последующее действие может быть начато. Если это не совсем так, то такая связь не должна существовать.

Рассмотрим, например, типичный набор действий, которые происходят, когда клиент заходит в ресторан. Сначала клиент заказывает столик, указывая размер своей компании. Администратор находит для них столик, возможно, просит подождать, а затем усаживает их. После того как клиент усажен, официант принимает заказ. Учитывая эту типичную последовательность событий, вы можете нарисовать каждое действие как предпосылку для следующего действия, как показано на рис. 5.2.


**Рис. 5.2.** Последовательность действий в ресторанной системе

Но действительно ли это правильное представление? Существуют ли ситуации, в которых официант может принять заказ, не усаживая гостей? Как насчет заказа на вынос, заказа из ресторана или клиента возле бара? Усаживание гостей не является обязательным условием для принятия их заказа. Это лишь действие, которое обычно происходит раньше и только в некоторых сценариях. Более точный анализ этой проблемы был бы таким, как показано на рис. 5.3.

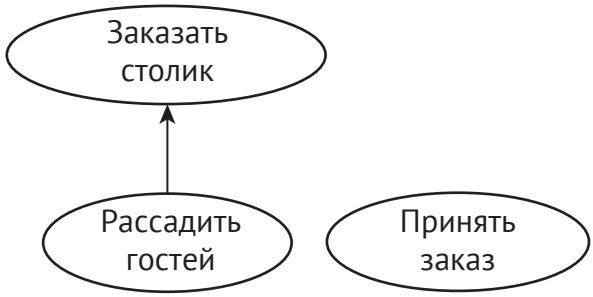


**Рис. 5.3.** Принятие заказа не зависит от усаживания клиентов

Помимо того что отношение предпосылки является строгим, оно переносит информацию вперед. Предпосылка предоставляет информацию для последующих действий. Нет необходимости дублировать информацию о предыдущем действии в последующее. Если последующее решение не опирается на информацию, содержащуюся в решении предпосылки, то, возможно, эта связь не должна существовать.

В предыдущем примере заказ стола включает число гостей. Это предоставляет информацию, которая ограничивает задачу по их рассадке. Отношение между рассаживанием гостей и заказом стола указывает на то, что мы знаем количество людей для рассадки. Пример диаграммы экземпляров представлен на рис. 5.4.

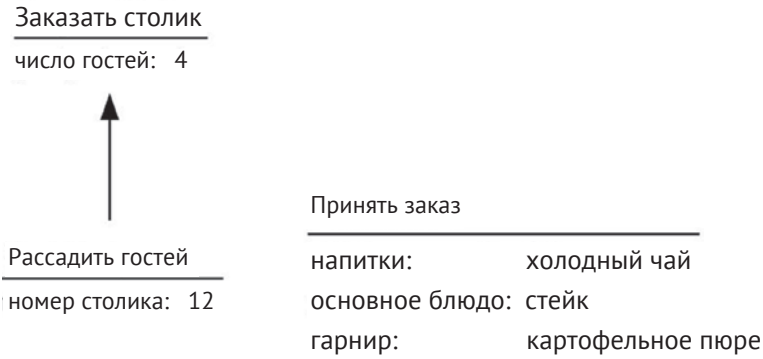


**Рис. 5.4.** Рассаживание гостей имеет доступ к информации об их числе из заказа стола

Заказ не нуждается в этой информации

Для выполнения действия «Принять заказ» не требовалось знать, где сидит клиент. Поскольку официант может общаться с клиентом, даже если это происходит по телефону или возле бара, он может выполнить действие «Принять заказ». Это еще один признак, который говорит о том, что рассадка гостей не является обязательным условием.

Когда сценарии использования большие, имеет смысл иметь несколько сложных типов отношений между ними. Крупный сценарий использования может включать в себя более мелкие, а общий сценарий может обобщать конкретные. Но когда действия выражены на атомарном уровне, важно только одно отношение. Диаграмма, изображающая последовательность действий в виде предпосылок, иллюстрирует возможные пути достижения цели. Каждое действие отображается *фактом* в исторической модели, а отношение предпосылки связано с *предшественниками*. Этот вид диаграммы может дать гораздо больше информации, чем введение требований в документ. Он может стать отправной точкой для действительно мощного анализа.

## Данные

Мы разбили сценарии использования системы на атомарные решения. Каждое решение не может быть разбито еще глубже. Оно принимается одним человеком, не имеет внутренних условий и не может быть прервано. Отношения между этими решениями являются отношениями *предпосылки*. Теперь мы можем начать более внимательно изучать данные, связанные с каждым решением.

Мы видели, как данные из решений предпосылок доступны для последующих решений. Эти данные доступны благодаря отношениям предпосылок. Это позволяет проанализировать, как накапливается информация, по мере того как мы соединяем большее количество решений в цепочку, обеспечивая более глубокую преемственность. Далее мы можем проанализировать содержание этой информации, чтобы обнаружить идентификаторы, кардинальность и изменение.

### Идентификаторы

Во многих структурах данных скрыты значения, идентифицирующие людей, объекты или другие сущности. Если вы обнаружили идентификатор, скрытый в некоторых данных, подумайте о том, чтобы извлечь его в отдельный факт. Замените поле отношением предшественника, указывающим на этот факт.

Продолжая анализ ресторанной системы, мы видим идентификатор в форме номера столика. Мы решаем поднять этот параметр до отдельного факта, переместив номер столика в него, чтобы он служил идентификатором.

Когда мы заменяем идентификатор в действии «Рассадить гостей» на отношение предшественника, то получаем диаграмму, изображенную на рис. 5.5.

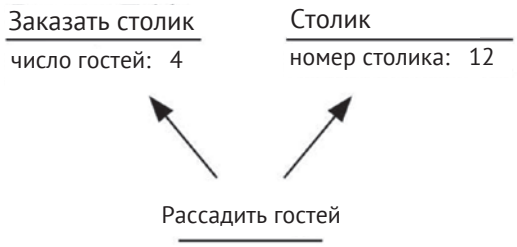


**Рис. 5.5.** Номер столика становится самостоятельным фактом

Возведение идентификатора в ранг факта создает новую интересную точку в приложении. Это достигает двух целей. Во-первых, позволяет наблюдать за всеми преемниками этого нового факта. Например, мы можем определить, какие лица сидели за столиком в течение определенного времени. Но, возможно, более важно то, что это дает нам основу для того, чтобы сказать больше вещей об идентифицированном объекте. В данном случае мы можем сказать, какой официант закреплен за столиком. Это помогает понять, кто несет ответственность за обслуживание гостей, а также сбалансировать будущую рассадку так, чтобы все официанты были загружены примерно одинаково. Результат показан на рис. 5.6.

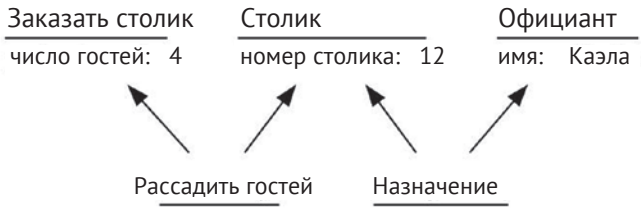


**Рис. 5.6.** Официант закреплен за столиком

По мере того как вы поднимаете идентификаторы до уровня фактов, вы обнаружите, что модель содержит смесь действий и сущностей. Действия были получены из сценариев использования, разбитых на атомарные решения. Сущности – это идентифицированные люди, объекты, места или понятия, о которых идет речь в этих решениях. Тем не менее диаграмма содержит только один вид отношений – предшественник. Это либо предварительное решение – то, которое должно было быть принято раньше, либо сущность, связанная с этим решением.

## Кардинальность

Продолжая анализ набора решений, мы можем определить части, которые имеют ноль, одну или много реплик. Это указывает на точки кардинальности[1], которые необходимо рассмотреть в модели. Часть, которая допускает только ноль или одну реплику, становится необязательным (?) предшественником. Часть, допускающая любое количество реплик, может стать либо множественным (*) предшественником, либо преемником. Разница между этими двумя понятиями связана с тем, могут ли новые части быть добавлены после факта.

Факт «Прием заказа» в предыдущем примере требует множественности. Изначально мы смоделировали его как единое целое с тремя полями, как показано на рис. 5.7.

[1] В программировании кардинальность – это мощность отношения, т. е. отношение количества сущностей-родителей к количеству сущностей-потомков. – *Прим. ред.*

Принять заказ

напитки: холодный чай

основное блюдо: стейк

гарнир: картофельное пюре

**Рис. 5.7.** Пример факта приема заказа, имеющего три поля

В данном примере имеет смысл обозначить каждое из этих полей как напиток, основное блюдо и гарнир. Данный конкретный заказ состоял из этих трех частей. Но в целом нам не нужно добиваться, чтобы в каждом заказе были именно эти три позиции. В некоторых заказах будут закуски, в некоторых – десерты, в некоторых могут быть два гарнира. Хотя типичный заказ может следовать шаблону или образцу, нет никакой пользы в том, чтобы ограничивать блюда в заказе конкретными категориями.

Более того, заказ предположительно предназначен для всей компании. Скорее всего, у нас будет несколько напитков, несколько основных блюд и т. д. Разные люди могут даже заказать закуски в качестве основных блюд. На данном этапе анализа важно спросить, имеют ли эти различия значение для решения, которое вы моделируете. Важно ли знать, кто заказал стейк, или официант просто спросит, когда они подойдут к столу? Важно ли моделировать момент подачи салата, или официант будет отслеживать каждое блюдо самостоятельно?

Решения, которые вы принимаете при анализе кардинальности модели, определяют те отношения, которые вы будете подчеркивать. Они будут отражать ценности владельца вашего товара. Одна из допустимых моделей ресторана позволяет гибко добавлять товары в заказ в любое время. Другая, но не менее допустимая модель будет фиксировать заказ так, чтобы его можно было контролировать на каждом этапе приготовления, доставки и оплаты. Если владелец товара ценит гибкость, а не контроль, то вы придумаете модель, показанную на рис. 5.8, которая позволяет добавлять преемников в любое время.


**Рис. 5.8.** Прием заказа позволяет добавлять несколько элементов заказа в качестве преемников

Факт всегда может иметь несколько преемников. Мы не указываем эту кардинальность на диаграмме, так как это подразумевается. Пример такой структуры фактов показан на рис. 5.9.

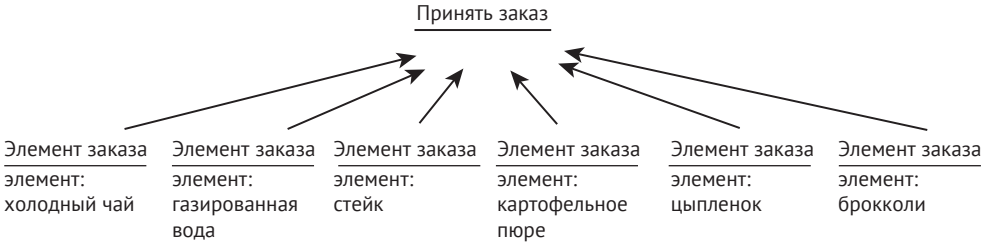


**Рис. 5.9.** Несколько элементов, представленных как преемники заказа

Решение моделировать позиции как преемников подчеркивает тот факт, что новые позиции могут быть добавлены в заказ в любое время. Если владелец товара принимает другой набор решений, то вы можете выбрать иную модель. Например, в ресторане быстрого питания заказ принимается целиком, готовится, а затем доставляется. Во время доставки нельзя вносить изменения. Это может заставить вас вместо этого сделать элементы заказа предшественниками факта «Прием заказа», как показано на рис. 5.10.

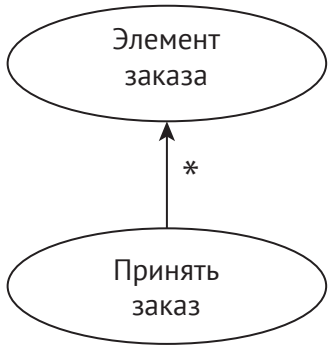


**Рис. 5.10.** Множественные элементы заказа являются предшественниками элемента «Прием заказа»

Поскольку элемент заказа является предшественником, мы *указываем* его кардинальность. Элементы заказа не могут быть добавлены постфактум. Звездочка (*) указывает на наличие нескольких элементов заказа, когда заказ фиксируется. Пример набора фактов, соответствующих этой модели, показан на рис. 5.11.

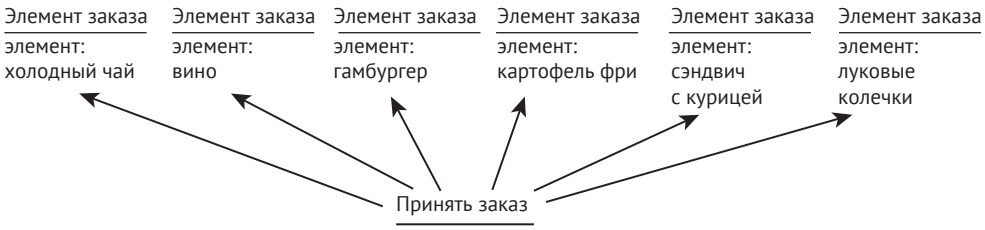


**Рис. 5.11.** Строгая организация элементов как предшественников, которые не могут быть изменены

Просто сказать, что в заказе много позиций, не достаточно, чтобы отразить нюансы этого процесса. Знание того, можно ли добавить товары впоследствии, является важной частью модели. Оно отражает ценности владельца товара и воплощается в возможности товара. Отражение их в модели является важным первым шагом на пути к анализу последствий этих решений.

## Изменение

Объекты, которые мы проанализировали и отобразили в модели, до сих пор поступали из трех различных источников. Некоторые из них были решениями, принятыми личностью на пути к достижению цели. Другие были идентифицируемыми сущностями, относительно которых были приняты эти решения. Третьи были меньшими частями этих решений и сущностей, которые появились в результате анализа кардинальности. В каждом случае эти объекты неизменны. Прошлое решение, личность или часть не изменится.

Это не всегда соответствует нашей интуиции. Когда мы смотрим на состояние системы, то представляем, что оно меняется с течением времени. Мы считаем это состояние изменчивым. Но то, что мы моделировали до сих пор, – это последовательность решений, которые *вызвали* видимую эволюцию изменения состояния. Модель представляет эти прошлые решения, она *не* представляет само состояние.

В какой-то момент нам нужно будет просто зафиксировать изменяемое свойство. Например, мы можем просто записать номер телефона человека. Нас не интересуют различные решения, которые приводят к заключению контракта на обслуживание с телефонной компанией. Эти решения были бы важны в другой области, но они не вносят вклад в ту область, которую мы пытаемся смоделировать. Мы просто хотим знать, как позвонить кому-то по телефону.

Моделирование изменяемого состояния следует рассматривать как последнее средство. Если вы можете представить свойство как следствие ряда бизнес-решений, то смоделируйте эти решения. Например, общая сумма чека в ресторане является следствием заказанных блюд, налогов и чаевых, а также любых скидок, которые может предложить ресторан. Такие свойства, как сумма чека, не должны быть представлены как изменяемое состояние. Используйте этот шаблон только для значений, которые не являются результатом моделируемого бизнес-процесса.

Например, название пункта в меню может быть смоделировано как изменяемое свойство. Нас не интересует бизнес-процесс, который приводит к появлению этого конкретного названия. Мы просто хотим задокументировать тот факт, что название может быть изменено. Мы делаем это путем записи свойства как факта, отдельного от сущности, которую он описывает. Это свойство имеет предшественника, который имеет ссылку на самого себя, называемую prior, и имеет кратность (*). Данная схема показана на рис. 5.12.

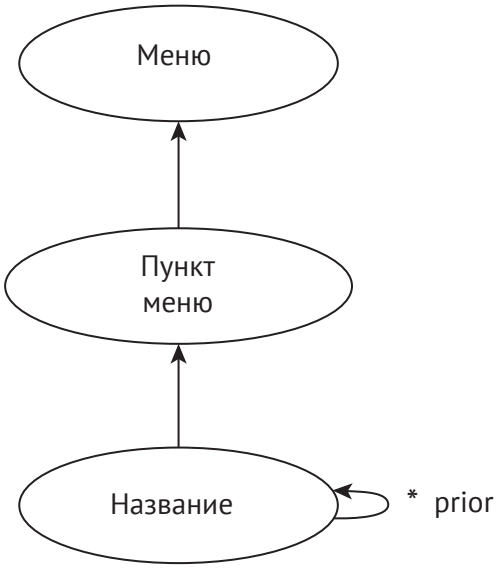


**Рис. 5.12.** Название пункта меню является изменяемым свойством

Некоторые изменяемые свойства имеют тенденцию изменяться как единое целое. Почтовый адрес, например, состоит из нескольких компонентов, таких как номер улицы, город и почтовый индекс. Предпочтительнее фиксировать эти составные значения как единое изменяемое свойство, а не как отдельные свойства, как на рис. 5.13. Это предотвращает неоправданную детализацию модели и в то же время документирует важную концепцию группировки данных.

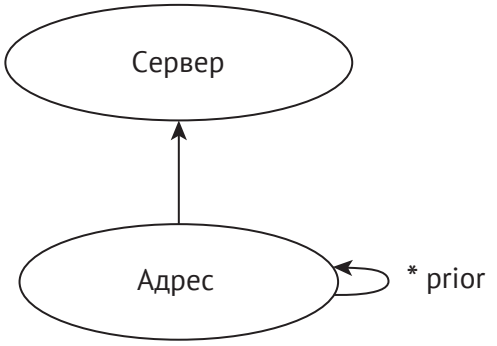


**Рис. 5.13.** Адрес – это единое изменяемое свойство, даже если он имеет много компонентов

Традиционная практика анализа при определении словаря данных не делает различия между свойствами процесса и изменяемыми свойствами. Словарь данных больше занимается перечислением различных свойств, которые могут быть прикреплены к сущностям, независимо от того, как они возникают. Но чтобы действительно проанализировать систему, нужно понять процессы, которые привели к появлению этих свойств. По возможности, эти процессы должны быть смоделированы как последовательные решения в рамках проблемной области. Но иногда изменяемое свойство – это просто изменяемое свойство.

## Представления

После моделирования решений, сущностей и свойств области у нас есть хорошее понимание ее базовой структуры. Модель, созданная до сих пор, отражает, как процесс развивается с течением времени. Каждое решение основывается на решениях прошлого в виде цепочки возрастающих знаний.

Чтобы принять решение, пользователь системы должен иметь доступ к этим знаниям. Задача системы состоит из двух частей – предоставить эту информацию и зафиксировать результирующее решение. Это новое решение становится новым фактом и вносит вклад в информацию, которую пользователи получат в будущем. Представления, которые система предлагает своим пользователям, могут быть выражены как функция их прошлых решений. Одна из задач аналитика заключается в том, чтобы описать эту функцию.

### Поиск точки старта

Чтобы извлечь данные из исторической модели, нам необходимо определить отправную точку. Мы не можем просто запросить всю модель в целом. К счастью, обычно у нас для этого есть несколько хороших кандидатов.

Большинство приложений требуют, чтобы пользователь вошел в систему. Как только они это делают, у нас есть отправная точка – сам пользователь. Войдя в приложение, пользователи будут переходить от страницы к странице. По мере этого они будут менять свое исходное место. Приложение будет предоставлять им информацию, основанную на этой точке модели. Отсюда они могут продолжить навигацию или принять решение, которое будет зафиксировано в модели.

Давайте продолжим создание модели ресторана, чтобы увидеть, как пользователь может быть отправной точкой. Официант входит в систему в начале своей смены. Оттуда он увидит все столики, за которыми он закреплен. Войдя в систему, официант определил начальную точку модели, которая показана на рис. 5.14.

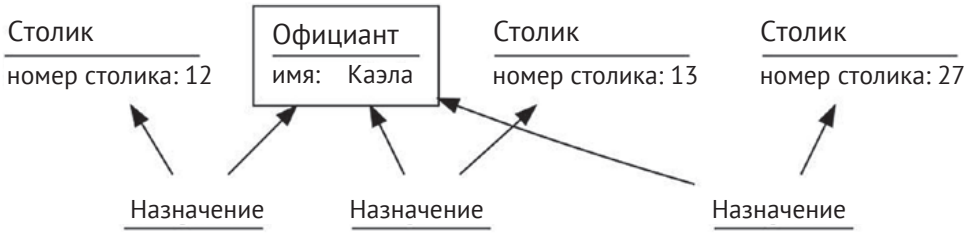


**Рис. 5.14.** Официант входит в систему, чтобы увидеть назначения своих столиков

Даже если приложение не требует от пользователя входа в систему, оно обычно имеет четко определенное начальное место. Как правило, это сущность верхнего уровня, которая определяет область действия приложения. Например, если у посетителя ресторана есть устройство за столиком для заказа напитков или закусок, ему, как правило, не нужно сначала входить в систему. Отправной точкой информации, которую он может искать, является меню. Он может перемещаться по пунктам меню, как показано на рис. 5.15.

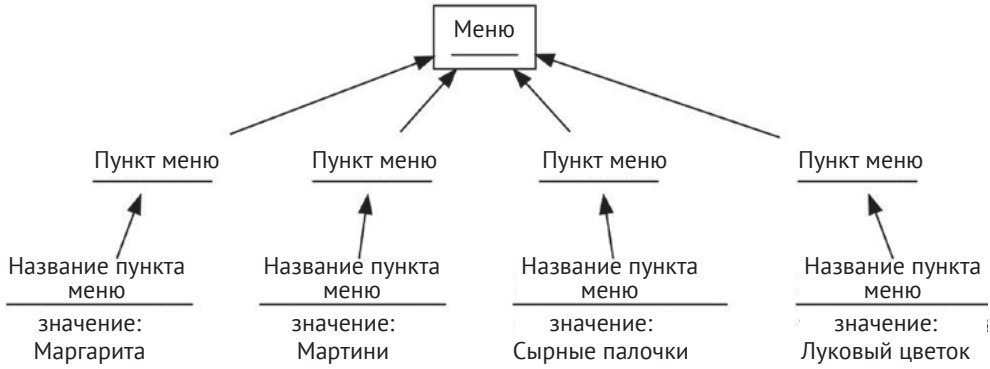


**Рис. 5.15.** Гость начинает с меню для поиска пунктов меню

Как только мы определили отправную точку, мы можем описать информацию, которая будет представлена пользователю. Для этого мы будем использовать схемы и запросы. Это, в свою очередь, приведет к уточнениям модели в итеративной внутренней спирали анализа.

## Аннотированные каркасы

Каркасные схемы – это мощный инструмент для передачи информации, которая будет представлена пользователю. Они применимы к любой системе, которая отображает информацию на экране. Они эффективны как для веб-приложений, так и для мобильных и настольных систем. И они становятся еще более мощными, когда аннотируются, чтобы показать, какую именно информацию следует отобразить.

Каждый каркас имеет начальную точку. Начальная точка четко обозначена и вводит остальные аннотации в контекст. Элементы внутри каркаса затем ан-

нотируются для документирования информации, которую они представляют. Эти аннотации принимают форму запроса от начальной точки. Например, каркас домашней страницы официанта показан на рис. 5.16.

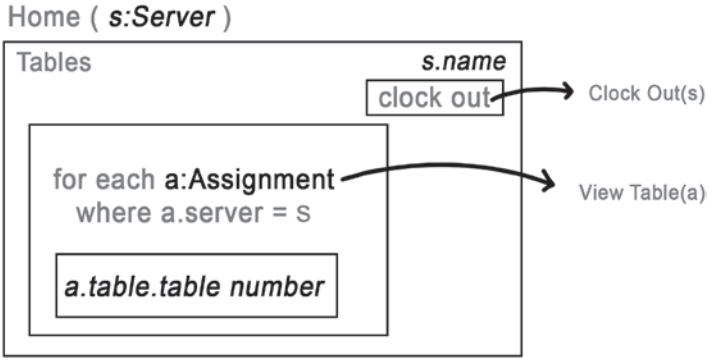


**Рис. 5.16.** Каркас домашней страницы официанта включает в себя аннотации к различным элементам

Когда пользователь переходит с одной страницы на другую, он выбирает новую отправную точку. Навигация появляется в виде линии, выходящей из каркаса, и несет с собой новую начальную точку для этого пункта назначения. Иногда это одна и та же точка (как в навигации к Clock Out), иногда она переходит в более узкий контекст (как в пункте View Table).

Аннотации не обязательно должны быть представлены непосредственно на каркасе так, как это показано на рис. 5.16. Они могут быть показаны в виде сносок, чтобы сделать каркас проще, или для того, чтобы оставить место для текста примера. Однако важным является то, что аннотации должны быть точными. Язык запросов Factual придает именно ту точность, которая необходима, чтобы поведение было однозначным, а предположения – явными.

## Удаление из списков

Исторические факты не могут быть удалены. Представления – это проекции исторических фактов. Однако любое представление, которое просто неограниченно разрастается, скоро станет бесполезным. Спецификации представления требуется способ удаления элементов из списков. Эта операция выполняется с помощью предложения `not exists`.

Рассмотрим администратора зала, принимающего решение о размещении гостей. Какая информация нужна для принятия этого решения? Нужно знать, сколько гостей ожидает, сколько столиков свободно, а также вместимость и размеры каждого из них. Имея под рукой эту информацию, он может решить несколько задач. Он может ввести запрос на столик, посадить гостей или указать, что ожидающие гости вышли. Эта информация может быть отображена в представлении, подобном показанному на рис. 5.17.

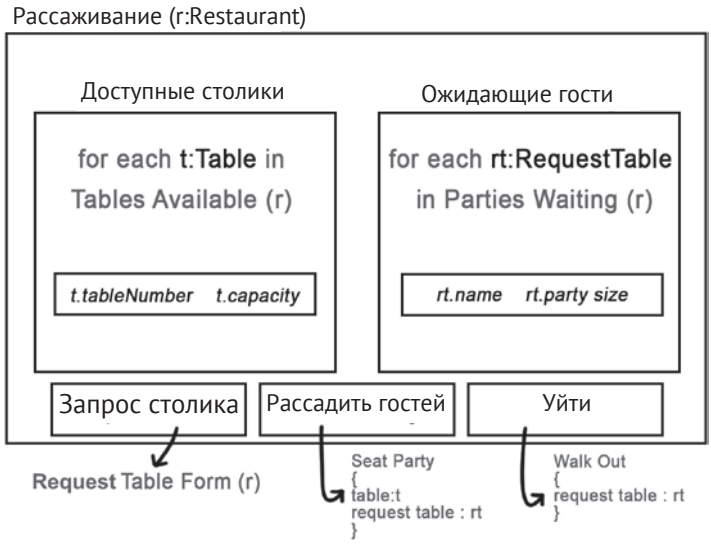


**Рис. 5.17.** Администратор видит свободные столики и ожидающих гостей и может предпринять соответствующие действия

Кнопка «Запрос столика» представляет администратору форму, в которой он может записать данные о прибывших гостях. Это инициирует навигацию. Кнопки «Рассадить гостей» и «Уйти», с другой стороны, сразу создают новые факты. Они записывают решения, связанные с выбранным столиком и запросом на столик. Эти решения будут влиять на информацию, отображаемую в представлении.

Но *как* именно эти факты будут влиять на представление? Какие события приводят к тому, что столик становится свободным? Что делает его недоступным? Ответы на эти вопросы становятся ясными, когда мы определим запросы, на которые ссылается схема, – «доступные столики» и «ожидающие гости». Начнем с первого.

Рассмотрим список столиков, за которыми можно рассаживаться. Мы не просто добавляем в него и удаляем из списка, происходит бизнес-процесс. Когда ресторан открывается, все столики свободны. Один из них удаляется из списка, когда за него садится посетитель. Он добавляется снова, когда столик освобождается. Мы можем смоделировать этот процесс с помощью единого запроса:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
    such that not exists s: SeatParty where s.table = t
      such that not exists b: BusTable where b.seatParty = s
}
```

Этот запрос говорит о многом в очень небольшом высказывании. В нем говорится, что рассаживание участников удаляет столик из списка доступных. Еще говорится, что освобождение столика возвращает его в список. Также

ясно, что нужно иметь отношение между столиком и рестораном, чтобы иметь отправную точку для запроса. Кроме того, нужно отношение между Bus Table (Освободить столик) и Seat Party (Рассадить гостей). Пересмотренная модель показана на рис. 5.18.


**Рис. 5.18.** Новые отношения, обусловленные необходимостью показывать доступные столики

Следуя аналогичной логике, мы можем проанализировать, что добавляют и убирают группы гостей, ожидающие столиков. Группа добавляется в список, когда она запрашивает столик. Она удаляется, когда их усаживают за стол. Группа также удаляется, когда она уходит. Это приводит нас к определению представления ожидающих гостей с помощью следующего запроса:

```
query partiesWaiting(r: Restaurant) {
  match rt: RequestTable where rt.restaurant = r
    such that not exists s: SeatParty where s.requestTable = rt
    and not exists w: WalkOut where w.requestTable = rt
}
```

Эти дополнительные факты и отношения еще больше уточняют нашу модель. Теперь можно увидеть, что запрос столика должен находиться в области действия ресторана. Мы также видим, как уход связан с запросом столика. Более полная картина представлена на рис. 5.19.

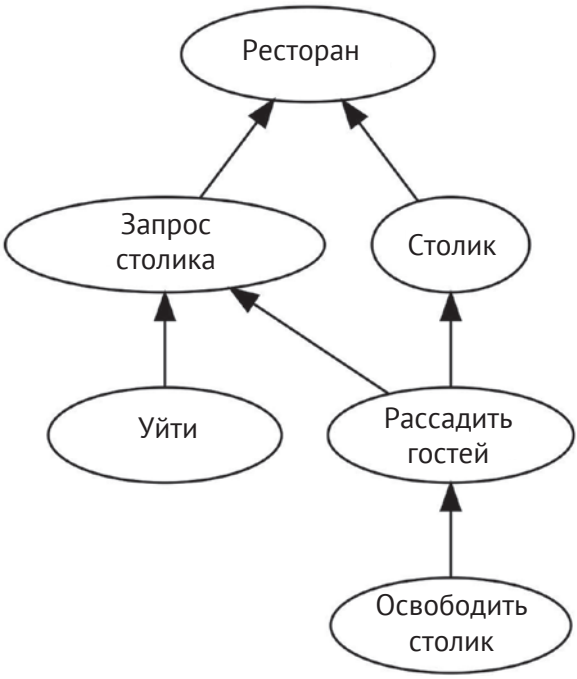


**Рис. 5.19.** Запрос столика происходит в пределах ресторана и может быть отменен с помощью ухода

Рассмотрев эти два запроса вместе, мы можем найти подразумеваемые требования. Мы можем увидеть, например, что две компании не должны сидеть за одним столиком в одно и то же время. Это можно заключить из того факта, что столик будет выбран из списка доступных (`tablesAvailable`), который не содержит ни одного стола с текущим «усадить компанию» (`SeatParty`). Более тонко запросы показывают, что при рассадке компании удаляется и компания, и столик. Это очевидно, потому что `SeatParty` появляется в пункте `not exists` каждого запроса.

Другие системы анализа потребовали бы от нас рассмотреть эти действия в терминах причины и следствия. Запрос столика *вызывает* добавление компании в список. Посадка *удаляет* компанию и столик. В итоге мы определяем конечный автомат, описывающий, как события изменяют состояние агрегатов. Легко забыть обновить состояние представления в ответ на событие, что приводит к неполной спецификации и ошибкам.

Точность языка запросов Factual раскрывает предположения о том, как состояние эволюционирует с введением новых фактов. Если бы требования к этим представлениям были выражены в прозе, было бы легко упустить эти предположения. Владелец продукта, знакомый с процессом управления рестораном, может даже не заметить предположения, которые он делает. *Конечно*, две компании не могут быть усажены за один и тот же столик. *Почему бы не* удалить и компанию, и столик при назначении?

Менее точная форма спецификации потребует от аналитика обнаружить эти невысказанные предположения и задавать вопросы. Если аналитик пропустит их, то разработчики могут столкнуться с нестандартными ситуациями. А если разработчики пропустят их, то тестировщики могут обнаружить дефект. Если мы требуем точности, то наш анализ выявит предположения, сделает поведение явным и позволит избежать потерь.

## Сотрудничество

По мере дальнейшего развития модели мы захотим обратить внимание на то, какие субъекты ответственны за те или иные решения. Разметка модели с указанием действующих лиц дает нам четкое представление о том, как система в конечном итоге будет использоваться в качестве инструмента для совместной работы. Люди будут работать вместе через систему. Она станет для них средством общения.

На диаграмме вариантов использования действующие лица рисуются внешними по отношению к системе в виде палочек. Стрелки указывают, за выполнение каких сценариев использования отвечают конкретные участники. Хотя это точное отображение – действующие лица находятся вне системы, – оно скрывает точки взаимодействия внутри модели. Гораздо понятнее провести линии ответственности внутри самой модели.

### Регионы

В своей работе в качестве аналитика я использовал несколько инструментов, чтобы указать, какой субъект отвечает за принятие того или иного решения. На доске или с помощью липких заметок я могу присвоить каждому действующему лицу свой цвет. В блокноте я могу сделать небольшую аннотацию в начале каждого факта. Но самый универсальный метод, который мы будем использовать здесь, – это разделить модель на регионы.

Регион – это область модели с ясно очерченными границами. Все факты в регионе – это решения, за которые отвечает один субъект. В системах моделирования процессов принято использовать «плавающие линии» для организации регионов ответственности. Хотя регионы в исторической модели не обязательно должны быть «плавающими линиями», эта практика вполне естественно переносится теми, кто к ней привык. При использовании «плавающие линии» обычно ориентированы вертикально, поскольку время идет вниз по странице.

Давайте переключимся на другую модель, чтобы более наглядно проиллюстрировать эту идею. Представьте себе систему, которая помогает организаторам конференций выбирать докладчиков, а затем составлять расписание для участников. В этой системе есть несколько субъектов, каждый из которых принимает различные решения. Организатор в первую очередь отвечает за проведение конференции и отбор докладов. Докладчик предлагает презентации для конференции. А слушатель выбирает, какие сессии посетить, оценивая их по окончании. На рис. 5.20 каждый из этих трех субъектов изображен как отдельный регион.

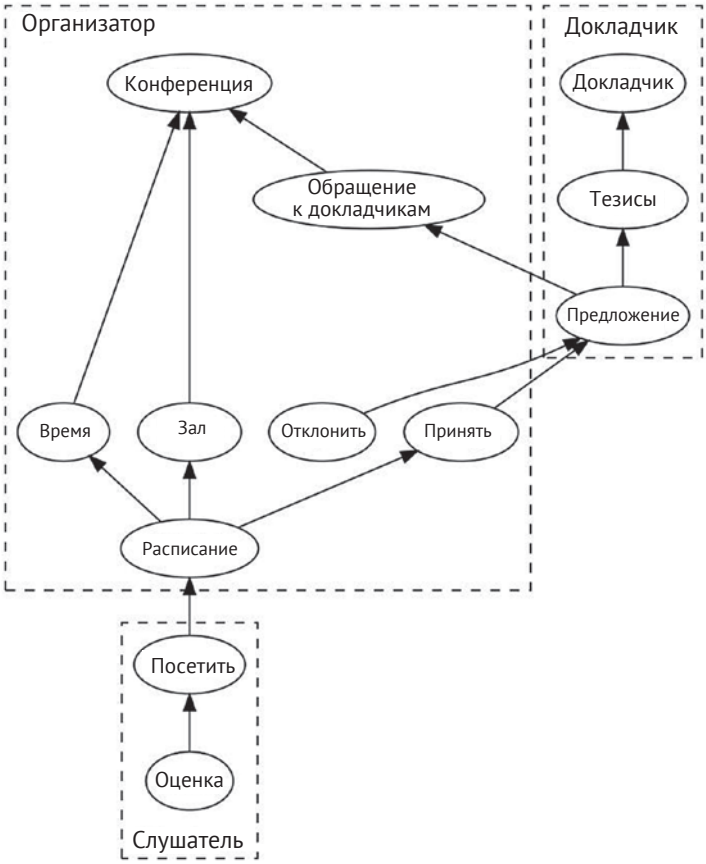


**Рис. 5.20.** Организаторы конференции, докладчики и слушатели имеют различную ответственность в рамках единой модели

Регионы организаторов и докладчиков представляют собой вертикальные плавающие плоскости, содержащие решения, за которые они отвечают. Процесс начинается с того, что организатор создает конференцию, а докладчик создает тезисы. Организатор публикует обращение к докладчикам, а докладчики подают предложения. Если предложение принято, то организатор планирует выступление. Слушатель появляется только в конце процесса, чтобы посещать и оценивать доклады. Вместо того чтобы показывать действия слушателя в плавающей дорожке, на рис. 5.20 они изображены в небольшой области. Этот выбор по-прежнему передает важную информацию – кто несет ответственность за каждое решение. Он также позволяет понять, когда линия пересекает границу ответственности.

## Пересечение границ

Когда регионы ответственности четко распределены, появляется важная особенность модели. Она имеет форму пересечения границы от одного региона

к другому. Решение, которое зависит от того, что находится за пределами контроля лица, принимающего решение, является важной точкой сотрудничества. Это указывает на то, что один субъект принял решение, а затем опубликовал его, чтобы другой субъект мог отреагировать на него. Такие точки в модели называются *опорными*.

Когда в модели появляется опорная точка, мы знаем, что информация, предоставленная одним субъектом, должна быть видна другим. На рис. 5.20 приведены два примера – рассылка обращения к докладчикам и публикация расписания. Организатор конференции отвечает за оба этих решения. В первом случае организатор публикует обращение к докладчикам в списках рассылки, социальных сетях, на веб-сайтах и через свою профессиональную сеть. Во втором случае он размещает расписание на своем веб-сайте и в мобильном приложении, а также раздает отпечатанные страницы. Аудитория каждой публикации обозначена хвостом стрелки – призыв к выступлению предназначен для докладчиков, а расписание – для слушателей.

В компьютерной системе опорные точки часто появляются как интерфейсы между взаимозависимыми подсистемами. Это особенно верно, если участники с каждой стороны являются членами разных организаций. В этом сценарии нередко опорная точка выражается в виде API. Получающий субъект – тот, что находится на хвосте стрелки, – делает API доступным бизнес-партнерам, чтобы они могли публиковать факты, подобные указанному в начале стрелки. Так будет, если организатор конференции использовал API для веб-сайта (call-forpapers – CFP) или платформы социальных сетей, чтобы опубликовать объявление для докладчиков.

Даже когда участники являются членами одной организации, опорные точки часто проявляются как связи между подсистемами. Например, если участники работают в разных отделах, нередко их системы взаимодействуют через передачу сообщений. В таких сценариях факт, находящийся во главе стрелки, будет иметь форму сообщения, опубликованного в топике или очереди. Это также может быть реализовано с помощью внутреннего API, общего подключения к базе данных или пакетного файла. Аналитику важно понимать, какие связи уже существуют и каковы возможности взаимодействия с унаследованными системами.

Опорные точки, ориентированные на потребителя, также являются важными характеристиками модели. Однако в развернутом приложении они часто принимают совершенно разные формы. Публичная опорная точка, такая как стрелка слушателя к расписанию, может просто отображаться в виде информации на веб-странице. Информация может быть доступна для поиска или приведена полностью. Обычно обновление страницы требуется для обновления представления, но для некоторых опорных точек веб-сайт или мобильное приложение может предоставлять уведомления. Аннотируйте модель с этими требованиями, чтобы команда разработчиков могла выбрать правильную реализацию опорной точки.

## Разговоры

В некоторых моделях между двумя субъектами возникает несколько опорных точек. Обычно опорные точки меняют направление и образуют цепочку фактов, путешествующих туда-сюда между ними. Так представляются разговоры, происходящие между двумя действующими лицами в системе. Один субъект публикует некоторую информацию, другой отвечает на нее, а первоначальный действует на основе этого ответа. Эти взаимодействия обычно обеспечивают наибольшую ценность системы и поэтому являются наиболее важными характеристиками для анализа.

Модель на рис. 5.20 содержит один пример разговора. После того как организатор конференции публикует призыв к докладчикам, докладчик отвечает на него предложением. Затем организатор отвечает «Принять» или «Отклонить». Это представляет собой сотрудничество между организатором и докладчиком. Организатор предположительно ведет такие разговоры со многими докладчиками одновременно. Каждый разговор несет в себе весь контекст – какой призыв к докладчикам, какое предложение, было ли оно принято или отклонено.

### Факты о публикации

Разговоры показывают, что между двумя субъектами должен происходить многоступенчатый процесс. Каждый из них знает о решениях, принятых другим участником разговора. Это относится даже к последнему решению – тому, на которое нет дальнейшего ответа. В предыдущем примере «Принять» или «Отклонить» является последним решением в этом разговоре. Несмотря на то что в этой модели у докладчика нет ответа, он осведомлен о принятом решении. Так обычно происходит в любом разговоре. Тем не менее модель не делает это явным. Используйте аннотированное представление или другую форму запроса на языке Factual, чтобы сделать это предположение явным.

В то время как первое сообщение в разговоре публикуется для широкой аудитории, последующие сообщения направлены конкретным участникам. Призыв к докладчикам публикуется в социальных сетях, но последующее принятие или отклонение предложения происходит через личное сообщение или электронную почту. Это сужение сферы охвата можно понять из структуры диаграммы. Докладчик является косвенным предшественником (предком) предложения. Это отношение указывает на то, что докладчик особенно заинтересован в ответах на данное предложение. Подобный интерес подразумевает, что системе необходима некоторая форма прямого уведомления, направленного докладчику. Мы формализуем эту концепцию и изучим механизмы ее реализации в главе 12.

## Взаимодействие подсистем

Когда субъекты, находящиеся по разные стороны разговора, используют различные подсистемы, каждая подсистема обычно должна иметь копию всего разговора. Независимо от того, находятся ли они в разных организациях или просто в разных отделах, их соответствующие приложения или микросервисы будут хранить свою собственную версию всех данных, которыми обмениваются. Модель показывает, какую информацию это может включать, – следуйте по стрелкам вверх от фактов, участвующих в разговоре. Все факты в этом восходящем конусе, называемом *транзитивным замыканием*, скорее всего, будут в той или иной степени дублироваться между системами. На рис. 5.21 показано, как этот набор может быть найден. Если вы обнаружите, что какая-либо информация *за пределами* транзитивного замыкания передается между системами, то у вас есть повод для беспокойства. Эта информация может измениться и потребует других разговоров для поддержания синхронизации.

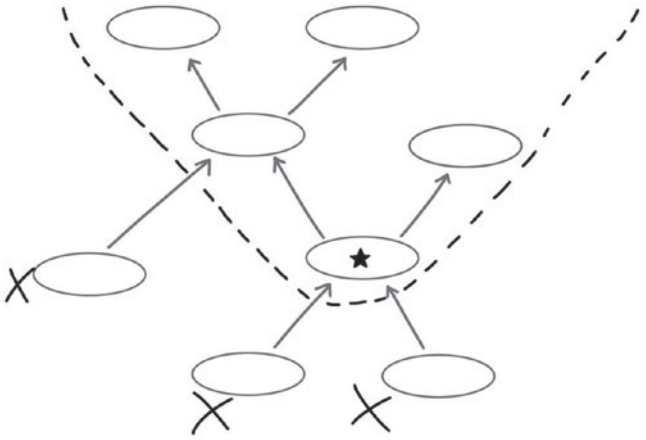

**Рис. 5.21.** Транзитивное замыкание включает всех предшественников и их предшественников. Оно не включает преемников

С каждой стороны разговора необходимо учитывать право собственности на данные и безопасность. Могут существовать требования, регулирующие перемещение данных за пределы юрисдикции страны. Такие нормативные акты и стандарты, как европейское Общее положение о защите данных (General Data Protection Regulation – GDPR), закон США о переносимости и подотчетности медицинского страхования (Health Insurance Portability and Accountability Act – HIPAA), Стандарт безопасности данных платежных карт (Payment Card Industry Data Security Standard – PCIDSS) или калифорнийский закон о конфиденциальности потребителей (California Consumer Privacy Act – CCPA), определяют порядок управления данными, хранения, защиты и доступа к ним. Эти руководящие принципы могут даже включать политику, согласно которой данные должны быть уничтожены. Хотя уничтожение данных, как правило, *не* допускается в рамках исторической модели, необходимо сделать исключения, чтобы учесть эти требования. К счастью, модель

помогает определить набор записей, которые должны быть удалены, – это конус, простирающийся *вниз* от объекта, который должен быть удален или забыт, как показано на рис. 5.22. Когда этот конус переходит из одного региона в другой, то системе требуются политики для обеспечения соответствия партнеров этим требованиям.

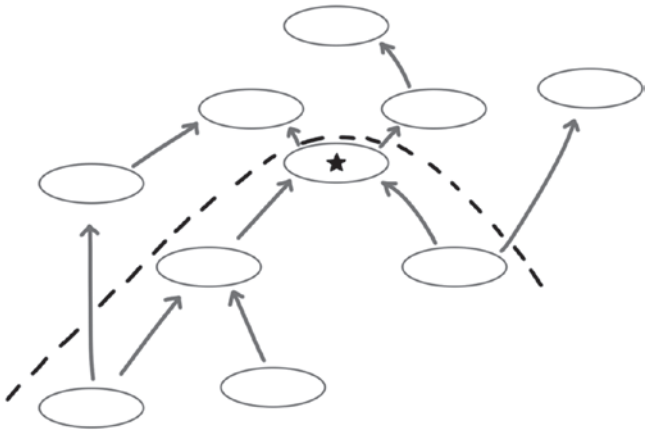

**Рис. 5.22.** Записи, удаляемые из исторической модели, включают всех преемников и их преемников

Опорные точки – это важные характеристики модели для анализа. Они представляют возможности для приложения взаимодействовать с внешними системами. Хотя работа аналитика не заключается в том, чтобы проектировать точки соединения, важно понимать, где они находятся и как будут проявляться. Различные формы интерфейсов будут по-разному ограничивать систему. Некоторые будут обеспечивать уведомления в реальном времени, в то время как другие потребуют запланированного опроса или периодического обновления. Как аналитик назовите места расположения опорных точек и соберите как можно больше информации об их ограничениях и требованиях.

## Допустимые упорядочения

При построении модели мы собрали граф связанных решений. Каждая стрелка указывает на то, что два решения должны быть приняты в определенном порядке – предшественник всегда происходит перед преемником. Но не менее показательными являются пары решений, между которыми нет стрелок. Это места, где порядок изменчив. Два факта могут иметь общего предка, но до тех пор, пока нет пути от одного к другому, эти два факта могут происходить в любом порядке.

Вспомним модель рассадки и назначения столиков в ресторане (рис. 5.23). Два факта «Рассадка гостей» и «Распределение» имеют общего предка – столик. И все же нет способа пройти по направлению стрелок от одного к другому. Это указывает на то, что компания может быть усажена за столик до или после того, как этот столик будет закреплен за официантом. Порядок этих двух решений не ограничен.

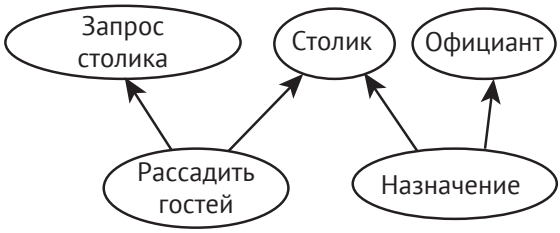


**Рис. 5.23.** Ни одна стрелка или цепочка стрелок не ведет от «Рассадки гостей» к «Распределению» или наоборот

Каждый из этих двух допустимых порядков – компания рассаживается до назначения столика или же столик назначается до того, как компания садится, – подразумевает разные требования. В первом случае, поскольку в момент получения официантом назначений столиков гости уже могут быть усажены, ему следует сообщить, какие столики уже заняты. А во втором, поскольку гости могут быть рассажены после этого, официант должен быть уведомлен, когда компания рассаживается за один из его столиков. Любой из этих вариантов допустим, но они вызывают разное поведение системы.

## Устранение условий гонки

В этом простом примере у нас есть только два допустимых порядка. Но в более реалистичной системе количество заказов может значительно возрасти. Поиск и учет всех возможных перестановок событий может оказаться сложной задачей. Аналитик может обнаружить, что по мере добавления каждого нового требования работа становится все сложнее. То, что начинается как простой список из нескольких возможностей, в конечном итоге превращается в лабиринт крайних случаев, каждый из которых пересматривается и расширяется с каждой новой функцией.

Для аналитика важно понимать, как система ведет себя при различных допустимых порядках событий. Описание каждой из перестановок поможет тестировщикам определить сценарии, которые необходимо изучить. Объяснение возможных вариантов поведения поможет разработчикам разработать код для крайних случаев. И самое главное, сравнение результатов различных допустимых упорядочений поможет выявить условия гонки. *Условие гонки* возникает, когда конечное состояние системы зависит от порядка, в котором происходили события. Чтобы избежать условий гонки, аналитик должен продемонстрировать, что все допустимые упорядочения сходятся к одному и тому же состоянию.

К счастью, запросы языка Factual, которыми мы аннотируем наши представления, предоставляют доказательство, которое нам необходимо. Когда мы выражаем поведение системы с помощью запросов Factual, а не серии причин и следствий, мы гарантируем, что состояние сходится к одному и тому же набору результатов, независимо от того, какое действительное упорядочивание имело место.

Кроме того, эти запросы также обеспечивают механизм для перечисления всех допустимых упорядочиваний. Они выявляют события, которые могут по-

влиять на информацию, представленную пользователю. Более того, они показывают нам, как реагировать на каждое из этих событий. Они указывают, как использовать предшествующие факты для обновления представления или уведомления пользователя. Из одной спецификации мы можем вывести правильное поведение для любого возможного порядка событий.

## Реагирование на различные допустимые заказы

Давайте вернемся к представлению, которое показывает официанту назначение столиков. Ранее мы только идентифицировали каждый назначенный столик. Но теперь расширим представление, чтобы оно также отображало размер любой компании, сидящей за этим столиком. Модифицированная аннотация показана на рис. 5.24.

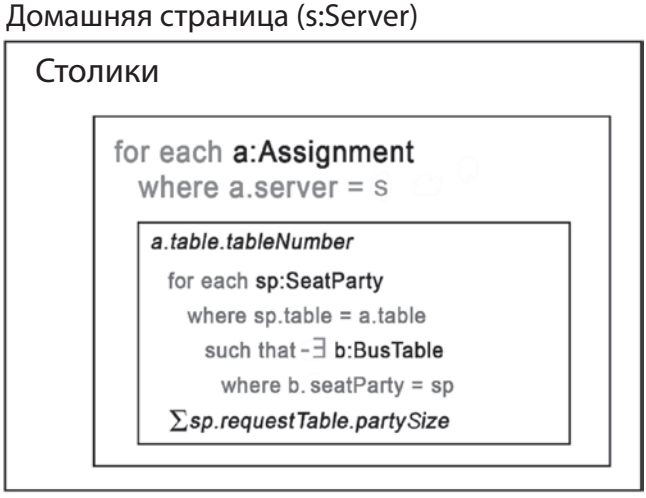


**Рис. 5.24.** Аннотированное представление домашней страницы официанта теперь показывает общее число гостей за каждым столиком

Объединение этих двух аннотаций дает нам запрос на все компании, обслуживаемые данным официантом. Давайте запишем его как запрос Factual, чтобы лучше понять, как это представление ведет себя.

```
query partiesAssignedToServer(s: Server) {
  match a: Assignment where a.server = s
  then sp: SeatParty where sp.table = a.table
       such that not exists b: BusTable where b.seatParty = sp

}
```

При прямом прочтении запрос дает инструкции по заполнению представления. Первое – найдите все назначения для официанта. Затем для каждого назначения найдите все компании, сидящие за столиком. Такая интерпретация запроса дает нам начальное поведение системы. Она выполняет запрос при загрузке представления, чтобы найти все сидящие компании.

Теперь давайте рассмотрим запрос, чтобы понять, как мы должны реагировать на изменения. Поскольку результаты запроса отображаются для официанта, мы хотим уведомлять его, когда результаты изменились. Давайте посмотрим на структуру запроса, чтобы определить, какие события могут вызвать такое изменение.

Запрос состоит из трех пунктов: один основан на назначении, второй – на «Рассадке гостей» и третий – на «Освобождении столика». Это подсказка о том, что любое из этих трех событий может привести к уведомлению. Мы можем разделить запрос по этим направлениям, чтобы определить, какие официанты будут уведомлены об этих событиях.

Первый пункт говорит нам, что представление будет меняться при появлении нового назначения. Когда менеджер назначает столик официанту, запрос будет обработан *для этого официанта*. Необходимо будет уведомить официанта. Назначение указывает нам, *какого* именно официанта нужно уведомить. Все посетители, уже сидящие за назначенным столиком, будут отображаться в этом представлении. Более формально: мы можем написать следующие фактические запросы, которые дадут нам официанта для уведомления и компании, которые были добавлены:

```
query serversToNotify(a: Assignment) {
  match a.server
}

query partiesAdded(a: Assignment) {
  match sp: SeatParty where sp.table = a.table
       such that not exists b: BusTable where b.seatParty = sp

}
```

Эта пара запросов описывает поведение системы, когда назначение происходит *после* «Рассадки гостей». Используя второй запрос, мы находим все существующие компании, которые все еще сидят за только что назначенным столиком. Если результат не пуст – это означает, что столик занят, – то первый запрос подсказывает нам, какого официанта нужно уведомить.

Но есть еще один допустимый порядок, о котором нам сообщает запрос `partiesAssignedToServer`. Второй пункт запроса подразумевает, что представление изменится, когда появится новая «Рассадка гостей». При рассадке компании мы должны уведомить официанта, прикрепленного к этому столику. Мы можем найти официанта, *инвертировав* запрос. Для полноты картины приведен также запрос, дающий добавленные компании.

```
query serversToNotify(sp: SeatParty) {
  match a: Assignment where a.table = sp.table
  then a.server
}

query partiesAdded(sp: SeatParty) {
  match sp
}
```

Обратите внимание, что второй запрос просто возвращает новый факт «Рассадка гостей». В нем нет дополнительных условий. В частности, в нем не упоминается факт «Освобождение столика», который обычно удаляет его из результатов. Причина этого в том, что данный запрос описывает реакцию системы на создание нового факта «Рассадка гостей». В момент создания не может быть факта «Освобождение столика» – еще не было времени для создания преемников. Формальное обоснование этого решения приведено в главе 9.

Используя инвертированный запрос, мы можем определить поведение системы, когда назначение происходит *до* «Рассадки гостей». Учитывая тот факт, что компания рассаживается, мы находим все назначения для этого столика. Каждое назначение дает нам официанта. Мы уведомляем каждого официанта, что у него есть новая компания – та, которая только что была рассажена.

Третий пункт – создание факта «Освобождение столика» – не приводит к добавлению компаний. Оно приводит к удалению компаний из поля зрения официанта. Если бы нашей целью было обновление представления, мы должны были бы включить этот сценарий в наш анализ. Но поскольку в настоящее время мы занимаемся уведомлением, то можем принять решение только об уведомлении официантов о компаниях *добавленных*, а не *удаленных*. Поэтому мы пропустим третий сценарий.

Сначала вам, возможно, придется убедить себя в том, что эта пара поведений дает все официантам, которые должны быть уведомлены о каждом событии, а также обо всех компаниях, о которых официанту необходимо узнать. Они охватывают все допустимые порядки событий и не позволяют ничему «провалиться сквозь трещины». Возможно, вы захотите проанализировать несколько различных сценариев, чтобы определить, почему это так.

Однако позже вы узнаете, что инвертирование запроса – это механический процесс. Его можно сделать для любого запроса, и он всегда дает полный и корректный результат. Вы узнаете, как выполнить этот процесс, в главе 9. На данный момент важно понять, что анализ системы с точки зрения запросов Factual выявит все допустимые порядки событий. Он точно описывает, как система должна вести себя в каждой из этих перестановок. И всегда сходится к одному и тому же результату.

## Последствия

Вы провели несколько итераций над своей моделью, и каждый проход уточнял факты немного больше, чем предыдущий. У вас есть четкое представление о том, как сценарии использования разбиваются на отдельные решения. Вы знаете, как эти решения связаны друг с другом. Вы определили субъектов, ответственных за каждое решение. Исходя из этого, вы определили опорные точки и разговоры в модели, которые поддерживают точки сотрудничества. Вы выразили информацию, которую видят эти субъекты, в виде запросов, и на их основе определили, какие события приводят к уведомлениям и обновлениям.

Теперь вы можете ответить на некоторые вполне реальные вопросы о возможностях вашей модели. На основе проектных решений, которые привели

к этому моменту, вы можете вывести ограничения, при которых будет работать полученное приложение. Вы можете определить последствия ваших решений по моделированию. Это поможет вам решить, удовлетворительны ли эти последствия, и, если нет, покажет вам, какой компромисс вам нужно сделать до того, как вы построите систему.

Последствия исторического моделирования не являются произвольными ограничениями. Они ограничивают наши возможности только теми вещами, которые можно легко сделать в распределенной системе. Если есть что-то, что модель не позволяет, то это потому, что реализация данной возможности в распределенной системе была бы запрещена. Это приведет к блокировке, потере автономности или снижению масштабируемости. Хорошо подумайте, нужна ли вам эта функция. Если да, то вам придется реализовать ее с помощью статической модели. Вы должны осознавать компромисс, на который идете, когда делаете это. Давайте подробно рассмотрим три из этих ограничений – индексы, количество результатов и порядок результатов.

## Индексы

Первым ограничением, которое вам нужно будет рассмотреть, будет то, как историческая модель может быть проиндексирована. Это повлияет на уникальность, навигацию и поиск. Что касается уникальности ограничений, учтите, что вы не можете гарантировать, что только один факт в распределенной системе имеет заданное значение для одного из своих полей, если только это не единственное поле, которое у него есть. Для навигации учтите, что вы не можете запрашивать факты, основываясь только на одном из их полей. Лучшее, что вы можете сделать, – это восстановить факт, учитывая все его поля, а затем запросить преемников. С другой стороны, поиск – это деятельность, которую лучше всего осуществлять вне исторической модели; определите, что должно быть доступно для поиска, и укажите, как будет получена информация.

### Ограничения уникальности

Ограничения уникальности часто являются весьма желательными и в то же время трудно реализуемыми в распределенной системе. Например, в системе учета дебиторской задолженности вы можете захотеть наложить ограничение уникальности на номер счета-фактуры. Если вы захотите, чтобы факт счета-фактуры имел дополнительные поля в дополнение к номеру счета-фактуры, то вы не сможете применить это ограничение в распределенной системе. Вам нужно будет собрать все счета-фактуры в одном месте и только потом проверить, что никакие два из них не имеют одинакового номера. Обратитесь к шаблону Исходящие в главе 8, чтобы узнать, как реализовать эту концепцию.

В исторической модели вся коллекция полей, включая предшественников, однозначно идентифицирует факт. Давайте воспользуемся этой предпосылкой для моделирования счета-фактуры с ограничением уникальности. Номера счетов-фактур не являются универсально уникальными, они уникальны только для одного поставщика. Поэтому у факта счета-фактуры (Invoice)

также будет предшественник поставщик (Vendor). Комбинация поставщика и номера счета-фактуры достаточна для идентификации счета-фактуры. Поэтому она также должна быть достаточной для создания факта Invoice, как показано на рис. 5.25.

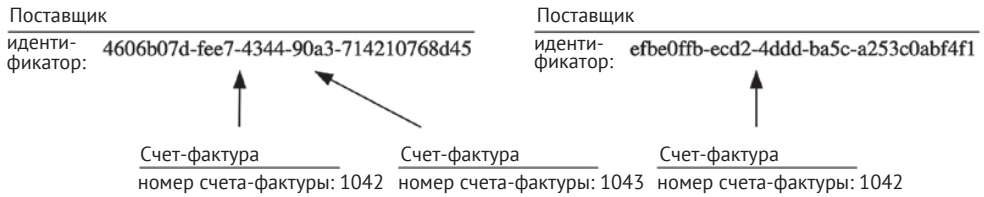


**Рис. 5.25.** Счет-фактура, имеющий уникальный номер счета-фактуры для каждого поставщика, не может иметь никаких других полей

Поэтому для моделирования счета-фактуры, имеющего уникальный номер, вам нужно будет, чтобы номер счета-фактуры был единственным полем в факте, не считая предшественника поставщика. Факт не может иметь никаких дополнительных полей, таких как адрес доставки (shippingAddress) или предшественник клиент (Customer). Добавление этих полей ослабило бы ограничение на номер счета-фактуры и позволило бы счетам с разными адресами или для разных клиентов иметь один и тот же номер.

Номера счетов-фактур являются примером *сгенерированного* уникального идентификатора. Необходимо не только обеспечить уникальность идентификатора, но новому счету-фактуре также должен быть присвоен новый номер. Даже если уникальный идентификатор изолирован, как на рис. 5.25, его генерация все равно должна происходить в одном месте. Сделайте пометку в своей модели, что определенное поле генерируется, и определите правила, по которым оно будет уникальным.

## Навигация

Далее рассмотрим влияние индексов на навигацию. Как мы видели ранее, факты являются отправной точкой для запросов в представлении. Когда пользователь переходит от одного представления к другому, он выбирает новый факт в качестве начальной точки следующего представления. Мы не можем делать запрос из поля, а только из факта. Если *хотим* начать с поля, то мы должны быть в состоянии построить соответствующий факт. Другими словами, такие начальные точки должны быть *единственными* полями в соответствующем факте.

Впервые мы заметили это, когда извлекали номер столика из факта `Seat Party`, чтобы создать факт `Table`. Благодаря этому выбору номер столика теперь можно использовать как индексированное поле. Система индексирует не поле, а скорее весь факт `Table`. Официанты могут перемещаться по модели данных, выбирая тот или иной столик, чтобы перейти к более детальным представлениям.

Выделение индексированного поля в отдельный факт является единственным способом инициировать исторический запрос по полю. С другой сто-

роны, в предложении SQL WHERE можно указать имя поля, даже если целевая таблица содержит больше полей. Но SQL работает на одной базе данных или кластере баз данных. Другими словами, он выполняется в определенном месте. Даже распределенный запрос с использованием механизма map-reduce, такого как Hadoop, масштабируется только на определенном кластере. Как SQL, так и не SQL-запросы в равной степени находятся за пределами более ограниченного исторического запроса. Сделайте пометку в модели, если требуется такой запрос, чтобы команда разработчиков могла реагировать соответствующим образом.

## Поиск

И наконец, давайте рассмотрим, как индексирование влияет на поиск. Поиск отличается от других индексированных запросов тем, что обеспечивает гораздо большую гибкость. Поиск может проводиться по части поля, как при поиске по префиксу, подстроке или SOUNDEX[1]. Поиск может также включать составные элементы, такие как булевы операторы или условия. Поисковые системы, как правило, имеют сложные и выразительные языки запросов.

Поиск по своей сути зависит от местоположения. Все записи, которые вы просматриваете, должны находиться в одном и том же месте. Это единственное «место» может быть кластером распределенных узлов, но мы считаем его одним местом по ряду причин. Во-первых, узлы, как правило, однородны – каждый из них был создан специально для того, чтобы стать членом кластера, и поэтому, как правило, работает под управлением одной и той же операционной системы, поисковой системы и хранилища данных. Во-вторых, узлы, как правило, расположены в одном месте. Они редко бывают географически разбросаны и почти никогда не управляются разными организациями. И в-третьих, количество узлов, как правило, известно или ограничено. Алгоритм распределенного поиска должен хорошо представлять, когда все узлы сообщили о себе, чтобы он мог объединить и представить результаты.

Статические хранилища данных лучше всего подходят для поиска. Хорошими примерами являются Elasticsearch или другие варианты Lucene. Эти хранилища данных оптимизированы для индексирования сложных записей или документов с использованием множества различных видов индексов. Некоторые индексы предназначены для числовых данных, позволяя фильтровать и сортировать диапазон. Другие индексы оптимизированы для фильтрации фрагментов текста с использованием операций с подстроками. Чтобы эффективно использовать эти системы, команда должна понимать, как документы будут поступать в поисковую систему, и различные способы, которыми они будут индексироваться. Именно здесь анализ может принести большую пользу.

Напишите запрос, определяющий, какие факты могут быть использованы для поиска. Ни один пользователь не будет выполнять этот запрос и просмат-

[1] Soundex – один из алгоритмов сравнения строк по их звучанию; устанавливает одинаковый индекс для строк, имеющих схожее звучание в английском языке. – *Прим. перев.*

ривать все результаты, их просто будет слишком много. Вместо этого сервис будет использовать этот запрос для подписки на факты, которые необходимо проиндексировать. Он преобразует факты в записи для поиска и добавит их в хранилище данных. Затем этот сервис может работать непрерывно, ища добавления, изменения и удаления документов, которые должны быть применены. Механизм идентификации этих событий подробно описан в главе 9. Документируйте исходный запрос, каждую из его инверсий и отображение этих фактов в документ, доступный для поиска.

## Ожидаемое количество результатов

Вторым ограничением, которое историческая модель накладывает на нашу систему, является количество результатов, которое мы можем ожидать от любого конкретного запроса. В запросе SQL вполне возможно написать предложение `WHERE`, которое, как вы знаете, будет соответствовать не более чем одной записи. Это происходит, когда фильтр основан на уникальном индексе или первичном ключе. Система управления базой данных обеспечивает соблюдение этих ограничений и неявно применяет к запросу предположение о единственном результате. Разработчики часто переносят это предположение в свой код и игнорируют все последующие строки, которые может вернуть запрос. Иногда они принимают дополнительные меры предосторожности в виде протоколирования или создания исключения, когда возвращается более одного результата. Но они почти никогда не корректируют свой код, чтобы позволить получить больше результатов.

Исторический запрос, однако, не может ограничить количество ожидаемых результатов. Причина та же, что и в предыдущем анализе, – отсутствие ограничений уникальности. Поскольку уникальность не может быть масштабируемо проверена в распределенной системе, исторические модели не дают такой гарантии. Это заставляет вас задуматься о том, что должно произойти, если запрос возвращает несколько результатов, в то время как вы ожидаете только один. Должны ли данные быть агрегированы? Должны ли множественные результаты быть перечислены? Должно ли представление выделять более одного результата как проблему, которую необходимо решить?

Иногда сочетание факторов делает крайне маловероятным, что запрос вернет более одного результата. Мы видели хороший пример этого в ресторанной системе. Вспомните, что представление администратора удаляло столик из списка, когда за него садился посетитель. Это было достигнуто путем получения списка из следующего запроса:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
    such that not exists s: SeatParty where s.table = t
      such that not exists b: BusTable where b.seatParty = s
}
```

Поскольку этот запрос является одним из источников для команды, создающей `SeatParty`, данное представление не будет создавать два факта `SeatParty` с одним и тем же столиком – по крайней мере, до тех пор, пока не будет

произведена первая рассадка. Таким образом, мы приходим к выводу, что запрос для компаний, сидящих за данным столиком, вернет не более одного результата.

```
query partiesAtTable(t: Table) {
  match s: SeatParty where s.table = t
       such that not exists b: BusTable where b.seatParty = sp

}
```

Однако модель не дает такой гарантии. Наша уверенность в этом ограничении основана только на поведении одного представления. Рассмотрим другие возможности, которые могли бы обойти это представление. Возможно ли, что другая подсистема может назначать компании? Могут ли два разных администратора открыть одинаковое представление зала в одно и то же время? Если мы допустим такую гибкость, то возможно появление двух рассаживаний за одним и тем же столиком. В представлении официанта мы решили разрешить множественные результаты путем суммирования размеров компаний (обратите внимание на Σ). Другими приемлемыми вариантами могли бы быть перечисление компаний или предупреждение официанта о потенциальной проблеме. Независимо от вашего выбора, вы должны признать, что каждый запрос может вернуть несколько результатов.

Однако есть одно исключение. Запрос, который полностью основан на предшественниках, ограничивается кардинальностью этих предшественников. Другими словами, запрос, который соответствует единственному предшественнику, всегда возвращает один результат. Если этот предшественник является необязательным, то он может не вернуть ни одного результата. Но если этот предшественник не имеет кардинально большого числа (*), то он вернет не более одного результата. Мы уже видели пример такого запроса при определении того, какого официанта следует уведомить о новом назначении столика.

```
query serverToNotify(a: Assignment) {
  match a.server
}
```

Этот запрос всегда будет возвращать ровно одного официанта. Подобные запросы часто возникают в результате инверсии запроса. Они редко возникают непосредственно в представлениях. Причина проста – эти запросы не могут меняться. Факты, на которых они основаны, неизменны. Никакой новый факт не приведет к тому, что существующий факт будет иметь другого предшественника.

## Отсутствие неявного порядка

Последнее ограничение распределенных систем – как показывает историческая модель – заключается в том, что результаты запросов не упорядочены. Даже если два узла достигли согласованности и возвращают один и тот же на-

бор фактов из запроса, они могут возвращать эти факты в совершенно разном порядке. Нам хотелось бы верить, что порядок появления фактов соответствует порядку, в котором они были созданы. Но в распределенной системе всегда существует вероятность того, что два решения были приняты параллельно, причем каждое из них не знало о другом. Когда это происходит, узлы, на которых эти решения были впервые получены, конечно, не согласятся с тем, в каком порядке они были приняты.

Неявно результаты запросов – это наборы, а не списки. Наборы не имеют порядка, только принадлежность. Нет ничего, что можно было бы вывести из порядка результатов, только наличие или отсутствие фактов. Программное обеспечение, однако, полно списков. Итерация коллекции по порядку – обычная особенность большинства языков. Мы даже не можем представить пользователю неупорядоченный набор, он всегда будет отображаться как список. Если мы итерируем результаты запроса, то обнаружим, что они расположены в *определенном* порядке. Мы просто должны быть осторожны, чтобы не зависеть от *этого* порядка.

Одна из целей распределенной системы заключается в том, чтобы различные узлы достигли согласованного состояния. Когда мы строим систему, основанную на принципах исторического моделирования, мы можем использовать CRDT (бесконфликтные реплицируемые типы данных), чтобы доказать, что мы достигнем согласованности, как показано в главе 4. Тип данных, который мы выбрали, был *набор*, а не *список*. Если бы мы выбрали списки, математика бы не сработала. Как бы то ни было, мы доказали, что наши наборы будут сходиться. Осталось доказать, что *проекции* этих наборов – информация, которую мы показываем пользователю, – также сходятся.

## Агрегаты

Проекции, которые мы представим, будут двух видов – агрегаты или итерации. Сначала рассмотрим агрегаты. Агрегат – это просто значение, вычисленное из упорядоченной коллекции. Сумма списка чисел – это агрегат. Максимум, произведение и стандартное отклонение также являются агрегатными функциями, которые могут быть вычислены из списка чисел. Важной особенностью всех этих агрегатных функций является то, что они *коммутативны*. Сумма будет одинаковой независимо от того, в каком порядке вы складываете числа. То же самое можно сказать о максимуме, произведении и стандартном отклонении. Коммутативные агрегаты полезны, потому что они игнорируют порядок списка.

Другие виды агрегатов не являются коммутативными. Один из примеров, который часто встречается, – это конкатенация строк. Имея список строк, принято добавлять их друг к другу, разделяя запятыми. Хотя этот агрегат полезен, он не является коммутативным, результат конкатенации строк зависит от порядка следования этих строк. В результате два узла могут вычислить разные результаты. Прежде чем использовать этот вид агрегата, необходимо сначала убедиться, что факты расположены в детерминированном порядке.

### Итерации

Это подводит нас ко второму виду проекции, который нам необходимо рассмотреть, – итерации. Итерации появляются в пользовательском интерфейсе в виде списков. Они также появляются в виде некоммутативных агрегатов, таких как конкатенация строк. Итерация применяется, когда имеется более одного результата, и подразумевает определенный порядок этих результатов. Поскольку результаты не имеют неявного порядка, итерация должна его установить.

Найдите какую-то особенность каждого факта, которая может определить порядок. Например, вы можете упорядочить объекты Столик (Table) по номеру столика. Прежде чем представить результаты пользователю, упорядочите их по этому полю. Пока каждый узел упорядочивает результаты одинаково, итерация (или некоммутативная совокупность) будет выглядеть последовательной.

Вы можете обнаружить, что хотите упорядочить факты в соответствии с изменяемым свойством. Будучи изменяемыми, эти поля не являются членами самих фактов. Они являются членами фактов-преемников, которые могут не присутствовать в каждом узле в одно и то же время. Когда это так, обязательно документируйте дополнительный запрос, который возвращает эти свойства фактов. Выразите порядок основных результатов в терминах проекции вторичных результатов. При этом помните, что вторичный запрос также может возвращать несколько результатов, и поэтому он тоже нуждается в коммутативном агрегате или детерминированном упорядочивании.

## Порядок создания

Наконец, есть один способ детерминированно упорядочить события в исторической модели по времени создания – записать время создания как поле. Это, безусловно, самый простой и прямой способ наложить детерминированный порядок на результаты. Однако это имеет два последствия. Во-первых, это требует, чтобы часы на узлах-источниках были синхронизированы. Если часы не синхронизированы, новые факты могут быть вставлены раньше старых. Во-вторых, что более важно, это заставляет время создания факта указывать в его идентификаторе. Два факта, которые были созданы в разное время, но в противном случае имели бы одинаковые значения полей, теперь будут рассматриваться как отдельные факты. Иногда это желательное поведение, а в других случаях – нет. Включайте время создания в факт только в том случае, если вам нужен дополнительный идентификатор, как обсуждалось в разделе «Идентичность» в главе 4.

Выбор, который вы сделаете при анализе проблемы, будет иметь последствия для окончательной реализации. Некоторые вещи, которые легко сказать, нелегко реализовать. Рассмотрения проблемы в терминах небольшого набора узлов или сотрудников недостаточно, чтобы раскрыть все эти предположения. Распределенные системы накладывают свой собственный уникальный набор ограничений.

Правила исторического моделирования намеренно ограничены, чтобы последствия решений становились очевидными. Вместо того чтобы разрешить уникальность или поиск по любому полю, исторические модели могут быть проиндексированы только по самому факту. Вместо того чтобы иметь возможность ограничить количество возможных результатов, исторические запросы всегда допускают множественность. И вместо того чтобы подразумевать порядок, исторические результаты требуют, чтобы проекции были либо коммутативными, либо явными. Внесите в свой анализ дополнительную информацию, необходимую для разъяснения ваших предположений, чтобы команда понимала все компромиссы, на которые им, возможно, придется пойти до того, как они напишут хоть одну строчку кода.

# Глава 6

# Переходы состояний

Среди наиболее мощных инструментов, доступных разработчику программного обеспечения, находится конечный автомат. Этот механизм, иногда изображаемый в виде диаграммы переходов состояний, описывает многоэтапный процесс. Автомат переходит из одного состояния в другое, по мере того как он встречает входные данные. Каждая единица входных данных определяет, по какой дуге графа переходит конечный автомат. Эта дуга определяет, как обрабатывается вход и в каком состоянии находится автомат для получения следующей единицы информации.

Данный инструмент является естественным выбором для решения таких задач, как анализ компьютерных языков и входных файлов. Спецификация формата обмена данными JSON[1], например, описана в терминах конечного автомата. Язык разбит на дискретные структуры, каждая из которых определяется диаграммой перехода состояний. Когда синтаксический анализатор ожидает увидеть определенную структуру, он переходит в состояние, изображенное на диаграмме. Например, диаграмма для анализа объекта может быть подобна показанной на рис. 6.1.

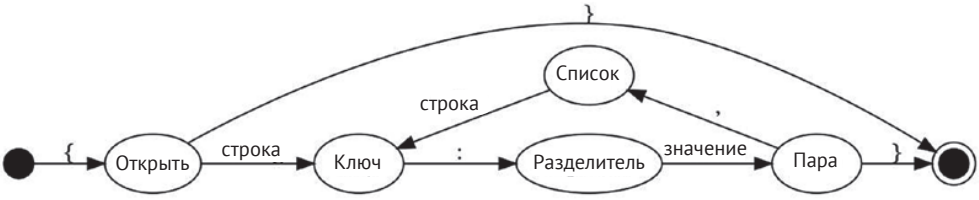


**Рис. 6.1.** Конечный автомат для анализа объекта JSON, как описано в формате ECMA-404

Диаграмма, как она представлена в спецификации, нарисована немного по-другому, но здесь мы называем каждое из состояний и помечаем ребра входом, который вызывает этот переход. Это обычный способ построения диаграмм переходов состояний. Состояние конечного автомата является изменяемым – оно изменяется по мере того, как он потребляет входные данные.

[1] The JSON Data Interchange Syntax. Standard ECMA-404. ECMA International. Second Edition, December 2017.

Учитывая выразительные возможности конечных автоматов, неудивительно, что многие люди применяют их в распределенных системах. Крис Паттерсон[1] (Chris Patterson), создатель распределенной системы MassTransit, рекомендует использовать конечные автоматы для управления сагами[2]. Джонатан Оливер (Johnathan Oliver), признанный в отрасли эксперт по распределенным системам, замечает: «Процесс лучше всего реализовать с помощью конечного автомата»[3]. Эти и другие лидеры в данной области продемонстрировали, как использовать конечные автоматы для управления бизнес-процессами, разделения длительных транзакций, защиты от дублирования сообщений и упорядочивания.

Мой опыт, однако, показал, что конечные автоматы являются плохим выбором для понимания и реализации бизнес-процессов в распределенной системе. Конечные автоматы хороши для реализации синтаксических анализаторов, но не распределенных бизнес-процессов. Все входные данные для парсера (синтаксического анализатора) поступают из одного источника – блока памяти или потока символов. Входные данные для бизнес-процесса, напротив, поступают из многих источников. Бизнес-процесс часто представляет собой бизнес-решения, принятые разными людьми с разными взглядами на систему. Парсинг происходит в одном месте, поэтому знать состояние парсера легко. Но бизнес-процессы происходят во многих местах одновременно, что затрудняет определение единого состояния системы.

Давайте рассмотрим некоторые проблемы, с которыми вы можете столкнуться при применении конечных автоматов в распределенных системах. Сначала мы попытаемся решить эти проблемы, используя инструменты, имеющиеся под рукой. Но в конце концов мы обнаружим, что проблема направляет нас к другому решению. Мы узнаем, как представление переходов состояний в виде неизменяемых фактов решает как аналитические, так и технические проблемы распределенных систем. А затем увидим, как перестроить как наше понимание, так и реализацию на основе этих новых методов.

## Множество свойств

Работая над системой управления цепочкой поставок, я столкнулся с первой из проблем – с переходом состояний при масштабировании. Мы использовали систему планирования ресурсов предприятия (enterprise resource planning – ERP) для создания приложения. Как и многие ERP-системы, та, которую мы использовали, была очень настраиваемой. Она позволяла разработчикам приложений определять свои собственные сущности, свойства и операции. Она также позволяла им определять конечный автомат.

---

[1] *Chris Patterson*. State Machine for Managing Sagas. Los Techies. 2009. https://lostechies.com/chrispatterson/2009/01/17/state-machine-for-managing-sagas/.

[2] В теории управления сага – последовательность связанных важных событий. – *Прим. ред.*

[3] Sagas with Event Sourcing. https://blog.jonathanoliver.com/cqrs-sagas-with-eventsourcing-part-i-of-ii/.

В ERP-системе разработчик определял состояния, в которые переходит объект по мере выполнения бизнес-процесса. Он определял, какие переходы из одного состояния в другое разрешены, а какие запрещены. Каждый переход изображался стрелкой между двумя состояниями и представлял собой шаг в процессе. Разработчики прикрепляли действия к этим шагам, чтобы настроить процесс.

Для простых конечных автоматов эта модель была вполне управляемой. Но по мере того, как система становилась более сложной, мы обнаружили, что умножаем количество новых функций на количество существующих состояний. Нам стало очевидно, что состояние элемента представляет собой более чем одно свойство. Иногда эти свойства взаимодействовали, а иногда – нет.

## Доставка и выставление счетов

Чтобы продемонстрировать проблему, давайте рассмотрим несколько более простой пример. Предположим, что мы построили систему, которая принимает заказы на товары на складе. Как только заказ размещен, отдел доставки выбирает товар и отправляет его клиенту. Тем временем отдел выставления счетов выставляет счет клиенту и получает оплату. Мы хотим, чтобы эти две операции выполнялись независимо друг от друга. Поскольку мы используем ERP-систему, которая определяет конечный автомат для каждой сущности, то разрабатываем граф состояний, который сочетает в себе эти две идеи, как показано на рис. 6.2.

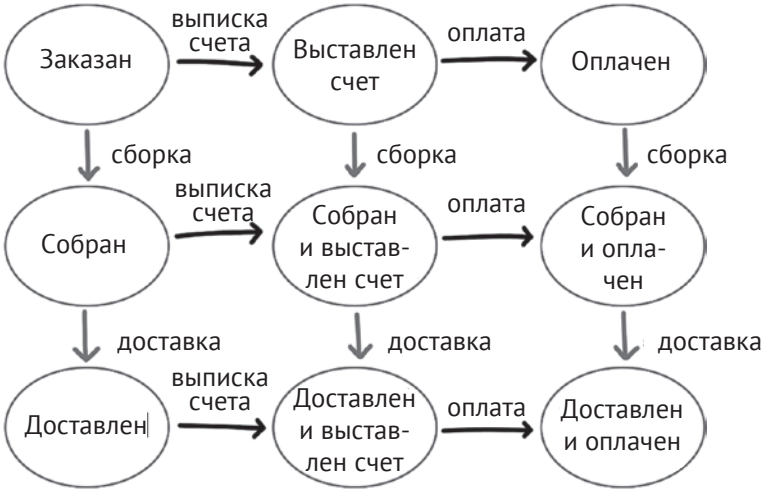


**Рис. 6.2.** Состояния заказа: выставление счета перемещается направо, а доставка – вниз

Этот конечный автомат позволяет отгрузке и выставлению счетов работать ортогонально. По мере того как отдел доставки собирает и отправляет заказы, состояние перемещается вниз по странице. А когда отдел выставления счетов оформляет счета и получает оплату, состояние перемещается по горизонтали. В конце концов, подобно таксисту, прокладывающему путь по Манхэттену, состояние достигает правого нижнего угла.

## Внедрение обратных заказов у поставщика

После того как система проработала некоторое время, компания осознает, что она вынуждена прекратить свою деятельность, когда у нее заканчиваются запасы. Чтобы исправить эту ситуацию, она расширяет свои операции, включив в них заказы у поставщика. Когда товара нет на складе, вместо отгрузки клиенту отдел доставки заказывает его у поставщика. Компания может либо получить его сама и затем отправить клиенту, либо попросить поставщика доставить его непосредственно к месту назначения. Измененная диаграмма перехода состояний показана на рис. 6.3.


**Рис. 6.3.** Продукты, отсутствующие на складе, были заказаны у поставщика. Заказ у поставщика не взаимодействует с выставлением счетов клиентам

Добавление функции заказа у поставщика потребовало от нас добавления трех состояний в диаграмму. В то время, пока продукт ожидают от поставщика, счет за него уже может быть выставлен и оплачен. Поэтому нам необходимо объединить состояние заказа с состояниями счета и оплаты. Поскольку эта новая функция не взаимодействует с существующей функцией выставления счетов, новые переходы имеют точно такие же действия, связанные с ними. Нет никакой разницы между доставкой заказа до или после его оплаты.

По мере добавления функций в эту модель мы обнаружим, что умножаем количество новых состояний на количество существующих состояний в независимых функциях. Было три состояния в процессе выставления счетов, поэтому одно новое состояние превратилось в три. Чем больше независимых процессов в рамках одного конечного автомата, тем больше становится этот комбинаторный взрыв.

## Отмены и возвраты

По прошествии некоторого времени компания решает, что она неправильно учитывает отмены и возвраты. Отмена заказа происходит до его отправки и может подразумевать возврат денег. Возврат, с другой стороны, происходит

после отгрузки заказа и требует увеличения запасов. Новая диаграмма перехода состояний показана на рис. 6.4.


**Рис. 6.4.** Отмена происходит до отправки, возврат – после

Отмена заказа почти полностью обрабатывается отделом выставления счетов. Поскольку продукт еще не отгружен, у отдела доставки практически нет работы. Они просто должны признать, что заказ перешел и что операция отгрузки больше не разрешена. Нет никакой стрелки из состояния «Отменен» с меткой «Отгрузка».

Возврат, с другой стороны, обрабатывается отделом доставки. Товар уже был отправлен, и поэтому им необходимо пополнить его запасы после получения. Только переход из состояния «Отгружено и оплачено» в состояние «Возвращено» затрагивает отдел выставления счетов. В этой ситуации они должны выдать возмещение.

При добавлении функций, которые взаимодействуют с существующими состояниями, мы часто можем избежать комбинаторного взрыва. Однако за это приходится расплачиваться сложностью. Мы должны теперь исследовать каждое существующее состояние, чтобы определить правильный ход действий, если в это время произойдет новая операция. Некоторые из этих переходов потребуют компенсирующих действий, а другие – нет.

## Параллельные конечные автоматы

Разработчик приложения, столкнувшись с этой проблемой, может найти ближайшее решение, имеющееся под рукой. Это решение не самое лучшее, но самое доступное. С помощью небольшой реорганизации алгоритма одно объединенное состояние может быть разбито на два или более параллельных состояний. В данном конкретном примере простое решение проблемы показано на рис. 6.5.

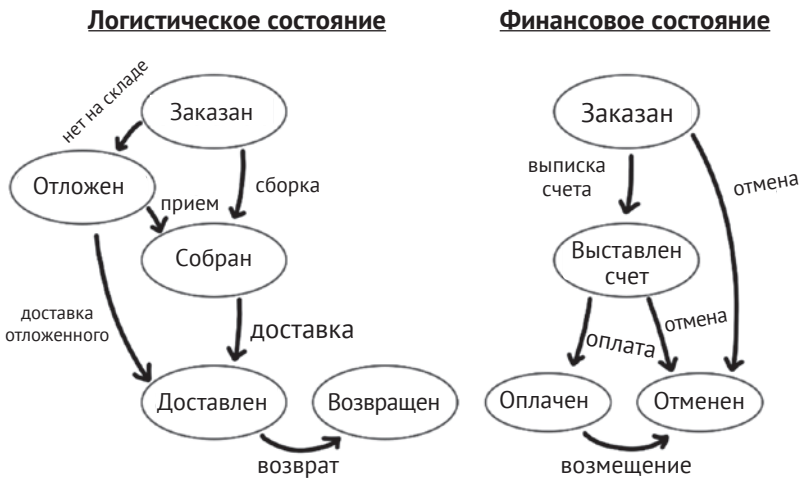


**Рис. 6.5.** Логистическое и финансовое состояния заказа разделены

Вместо одного состояния у объекта заказа есть два. Состояние логистики отслеживает процесс, как его видит отдел доставки. Финансовое состояние фиксирует процесс отдела выставления счетов. В той мере, в какой эти процессы могут протекать независимо друг от друга, эти два конечных автомата работают параллельно.

Однако когда эти процессы взаимодействуют, операция должна быть чувствительна к обоим состояниям одновременно. Если заказ возвращается, мы ожидаем, что состояние логистики начнется с состояния «Отправлен». Операция переведет его в состояние «Возвращен». Однако она также должна учитывать, является ли финансовое состояние «Оплачено». Если да, то она должна выдать возврат. В любом случае она должна перевести финансовое состояние в «Отменен», чтобы с клиента не взималась плата.

Параллельные конечные автоматы избавляют от комбинаторного взрыва, который возникает, когда одно состояние моделирует независимые процессы. Они также вносят упрощение, потому что не нужно рассматривать крайние случаи для каждой отдельной комбинации. Они находятся под рукой у разработчика приложений, использующего изменение для построения системы. Но они не являются идеальным решением, так как оставляют нерешенными несколько проблем. Одна из таких проблем связана с агрегатами – объектами со многими дочерними элементами.

## Много дочерних элементов

После решения проблемы объекта, имеющего много свойств, мы все еще остаемся с проблемой того, что у объекта много дочерних элементов. С одной стороны, каждый дочерний элемент может иметь свое собственное состояние. Но с другой – состояние родительского объекта может зависеть от состояния дочерних объектов. Это приводит к взаимодействию между конечными автоматами, даже когда они создаются и уничтожаются динамически.

Родительский конечный автомат отслеживает общий процесс. В определенный момент процесс разветвляется. Каждый дочерний объект должен проходить дочерний процесс параллельно. Только после того, как все дочерние процессы завершены, мы разрешаем родительскому процессу продолжить работу. После того как основной процесс продвинулся вперед, добавление дочернего может остановить прогресс и переместить родительское состояние назад. Удаление этого потомка должно снова продвинуть состояние родительского процесса вперед.

Описать все эти взаимодействия в виде переходов состояний становится все сложнее, поскольку каждый новый сценарий порождает все большую паутину побочных ситуаций. Наше первое побуждение – создать механическое решение: «Когда в этом состоянии происходит вот это, сделай это и перейди в это состояние». Такая логика быстро становится сложной для осмысления. Паутина побочных ситуаций становится укрытием для дефектов. Мы разрубаем этот гордиев узел, представляя состояние не как механический набор переходов, а как декларативную функцию.

## Отслеживание проблем в программном обеспечении

Один из моих клиентов использует популярное решение по отслеживанию проблем для управления исправлениями ошибок, функциями, пользовательскими историями и других изменений в своем программном обеспечении. Подобно ERP-системе, эта программа позволяет пользователям настраивать свой рабочий процесс, определяя состояния и переходы. По умолчанию ошибка может начинаться при сортировке, переходить через состояния «В работе» и «В тестировании» и, наконец, перейти в «Готово». Однако мой клиент изменил этот рабочий процесс. Он должен убедиться, что все изменения были проверены на соответствие нормативным требованиям. Поэтому он вставил «Ожидание рассмотрения» до того, как изменение перейдет в состояние тестирования. Измененная диаграмма перехода состояний дефекта показана на рис. 6.6.


**Рис. 6.6.** Диаграмма перехода состояний для дефекта программного обеспечения, измененная для обеспечения рассмотрения изменений

Функции проходят через аналогичный рабочий процесс. Каждое изменение программного обеспечения мотивировано наличием функции или исправлением ошибки. Благодаря этому мой клиент может быть уверен, что каждое изменение прошло через этап рецензирования. Это решает проблему, которая стояла перед ним изначально, – обеспечение соответствия требованиям. Однако это создает другие проблемы.

Каждое изменение программного обеспечения – это фиксация в системе контроля версий. Для исправления ошибки может потребоваться несколько коммитов. Обычно разработчики отделяют рефакторинговые изменения от

исправлений, чтобы сделать свое намерение более ясным. И тем не менее вся ошибка рассматривается как единое целое. Поскольку эта диаграмма перехода состояний реализована в коммерческом инструменте для отслеживания проблем, мой клиент ограничен в том, как он может разбить объекты на части. Инструмент не позволяет ему представлять состояние коммита как относящееся к проблеме. И поэтому он фиксирует рассмотрение на объекте «Ошибка» (Bug), а не на «Изменении» (Commit).

## Дочернее состояние

Чтобы решить эту проблему без изменения системы отслеживания проблем, мой клиент использует другие сторонние инструменты. Один из них – система обзора изменений, которая позволяет разработчикам комментировать отдельные строки каждого коммита. В рамках этого инструмента разработчики и рецензенты обсуждают мотивацию, рекомендации и возможные исправления кода. В конце этого процесса рецензент изменяет состояние коммита. Диаграмма перехода состояний, применяемая на уровне коммита, показана на рис. 6.7.


**Рис. 6.7.** Диаграмма переходов состояний коммита, реализованная в отдельном инструменте

Сочетание этих двух инструментов требует дисциплины. Разработчик переводит инструмент отслеживания проблем в состояние «Ожидание рассмотрения», а затем приглашает рецензентов присоединиться к обсуждению в системе рассмотрения изменений. Рецензенты перемещают коммиты в свои индивидуальные рабочие процессы, запрашивая дополнительные коммиты для решения возникающих проблем. Только после того, как все коммиты в ветви приняты, код объединяется, а дефект переводится в состояние «Тестирование».

## Составные диаграммы перехода состояний

Предполагая возможность объединения этих двух инструментов в соответствии с нашими потребностями, разработчик может объединить две диаграммы перехода состояний. Ошибка будет перемещаться через родительский конечный автомат, а каждый коммит будет проходить через дочерний конечный автомат. Составная диаграмма перехода состояний показана на рис. 6.8.

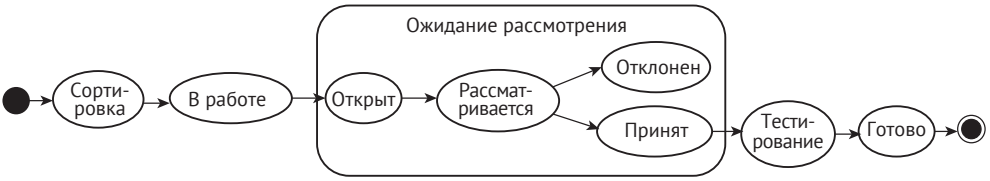


**Рис. 6.8.** Составная диаграмма перехода состояний позволяет родительскому конечному автомату продвигаться вперед только после того, как все дочерние достигнут конечного состояния

Чтобы реализовать эту диаграмму механически, нам нужно рассмотреть, как состояние родительского объекта взаимодействует с состоянием дочерних объектов. Если родитель находится в состоянии «Ожидание рассмотрения» и ребенок принят, проверьте, все ли остальные дети также приняты – если да, переведите родителя в состояние «Тестирование». Если из ветви удаляется отклоненный ребенок, проверьте, был ли он последним, и если все остальные приняты, переместите родителя вперед. Если родитель находится в состоянии «Тестирование» или «Готово» и добавляется новый ребенок, переместите родителя обратно в «Ожидание рассмотрения».

Учитывает ли этот механизм все возможные сценарии? Трудно сказать. Бремя доказательства лежит на разработчике, внедряющем решение. Проверка всех краевых случаев зависит от тестировщика. И по мере добавления новых состояний их количество растет.

## Декларативная функция состояний

Вместо механического решения мы можем определить декларативное. Декларативные решения поддаются доказательству гораздо легче, чем механические. Довольно легко рассмотреть декларативное утверждение и увидеть, все ли возможные условия перечислены. Функция вычисляет общий рабочий процесс на основе состояний отдельных компонентов.

Мы можем описать рабочий процесс ошибки как функцию родительского и дочернего состояний. Для этого мы воспользуемся *универсальным квантификатором*. Это просто модная фраза, означающая «для всех». Родительские состояния «Сортировка» и «В работе» отображаются непосредственно в рабочий процесс. Но разница между «Ожидает рассмотрения» и «Тестирование» зависит от дочерних состояний. Ошибка находится в тестировании, если *для всех коммитов в ветви* коммит принят. В противном случае он ожидает рассмотрения.

Эта логика может быть написана декларативно, как в следующем псевдокоде:

```
workflow (bug) =
   if bug.state = Triage
      then Triage
   else if bug.state = InProgress
      then InProgress
   else if not for all commit in bug.branch, commit.state = Accepted
      then AwaitingReview
   else if bug.state = InTest
      then InTest
else Done
```

Проверка на то, что все коммиты приняты, останавливает рабочий процесс от продвижения дальше «Ожидания рассмотрения». Не важно, перешло ли родительское состояние дальше, любой незавершенный дочерний компонент задержит рабочий процесс. Такое декларативное описание означает, что нам не нужно писать механизм, который обрабатывает все возможные случаи. Если новый коммит добавляется, пока ошибка находится в тестировании, универсальный квантификатор заставляет это булево выражение вернуться к ожиданию рассмотрения. А если коммит удаляется из ветви, универсальный квантификатор повторно оценивает и позволяет рабочему процессу продвигаться дальше. Без декларативного выражения разработчику пришлось бы писать код для таких крайних случаев и доказывать, что были изучены все возможные сценарии.

Декларативная функция находит крайние случаи, вызванные взаимодействием между родительским и дочерними состояниями. Написание функции в терминах состояний – это короткий шаг от управления переходами между состояниями с помощью изменения. Однако это все еще не идеальное решение. Теперь, когда в нашем распоряжении есть несколько примеров и мы рассмотрели несколько возможных решений, можем погрузиться в более сложный вопрос условной проверки. Это окончательно приведет нас к отказу от решений, основанных на изменяемости.

## Условная проверка

Когда объекты проходят через процесс, они не просто меняют состояние. Они также накапливают данные. Когда мы переводим объект из одного состояния в другое, то хотим записать эти новые данные. В идеале мы выполняем обе операции в рамках одной транзакции, чтобы наличие данных соответствовало текущему состоянию.

В зависимости от состояния объекта поля данных могут быть пустыми или требовать значения. Проверка полей данных зависит от состояния объекта. Системы типов наших баз данных и инструменты языка программирования обычно не учитывают такую условную проверку. Поэтому мы вынуждены ослаблять объявленные типы, чтобы компенсировать это.

Мы рассмотрели два примера – выполнение заказов и отслеживание изменений в программном обеспечении. В каждом из этих примеров мы находили решения проблем перехода между состояниями по мере их возникновения. Но мы не анализировали другие поля объектов. Давайте рассмотрим эти поля и посмотрим, есть ли у нас условная проверка.

## Допустимость неопределенного состояния

В примере с выполнением заказа мы хотим записать информацию о заказе и каждом элементе, который он содержит. Для заказа мы записываем клиента, адрес доставки и адрес выставления счета. Для каждого товара мы фиксируем артикул, текущую цену и количество. Когда мы отправляем товар, мы также хотим указать номер отслеживания. Где он должен быть?

Сначала может показаться, что номер отслеживания должен быть частью заказа, как показано на рис. 6.9. Когда заказ размещен, у нас еще нет номера отслеживания, поэтому мы позволяем этому полю быть пустым (фактически иметь значение NULL. – *Прим. ред.*). Оно заполняется только при отправке заказа. И поэтому отправленный заказ будет иметь непустой номер отслеживания. Установка состояния «Отправлен» и заполнение номера отслеживания произойдут в одной транзакции.

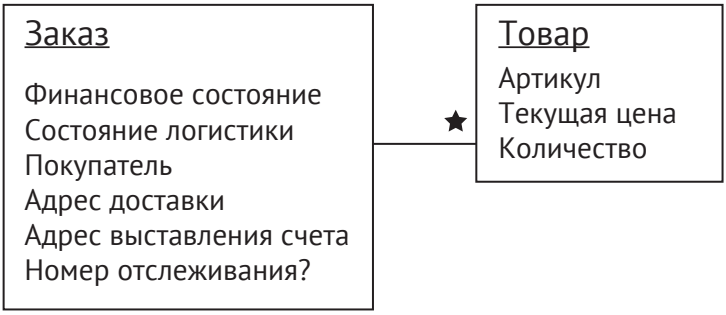


**Рис. 6.9.** Номер отслеживания – это обнуляемое поле в заказе, допускающее пустое значение, в котором также хранится состояние логистики

Но, возможно, состояние логистики должно быть частью товара, как на рис. 6.10. Каждый товар может быть заказан, выбран, отправлен и возвращен по отдельности. Если это так, то имеет смысл поместить номер отслеживания на товар, а не на заказ. Опять же, когда товар добавляется в заказ, у него нет номера отслеживания. И поэтому это поле не имеет значения до тех пор, пока товар не будет отправлен.

Независимо от того, размещаем ли мы номер отслеживания на заказе или на товаре, мы сталкиваемся с условной проверкой. Номер отслеживания должен быть пустым, если состояние логистики – «Заказан», «Задержан» или «Собран». Он не должен быть пустым, если состояние логистики – «Доставлен» или «Возвращен». Как только мы пройдем определенный этап рабочего процесса, номер отслеживания должен будет присутствовать. Проверка этого поля зависит от состояния объекта.

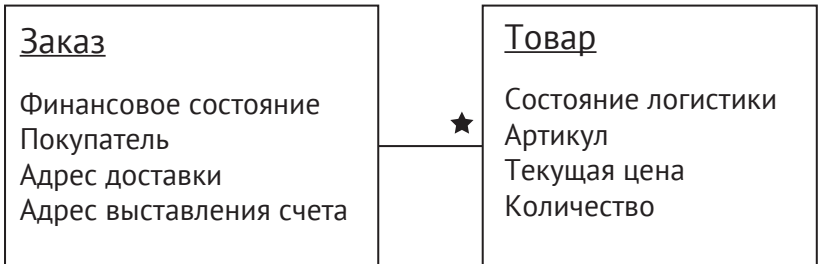


**Рис. 6.10.** Номер отслеживания является полем с допустимым пустым значением в товаре, куда мы перенесли состояние логистики

## Циклы в изменении состояния

Давайте рассмотрим еще одну модификацию, которую мой клиент внес в систему отслеживания изменений в программном обеспечении. Система позволяет пользователям добавлять пользовательские поля к ошибкам и функциям. Мой клиент добавил поле для отслеживания тестировщика, который проверяет изменение. Когда ошибка или функция впервые создается, это поле не заполняется. После проверки тестировщик вносит в него свое имя и изменяет состояние на «Готово». Заполнение поля и изменение состояния выполняются в одной и той же транзакции.

Это новое поле завершает картину. В коммите есть постоянная запись о том, кто внес изменение. Из системы рецензирования мы знаем, кто проверил каждый коммит. А теперь это новое поле говорит нам, кто протестировал исправление. Но это поле также создает проблему.

На диаграмме перехода состояний есть стрелка, которую мы раньше не рисовали. Тестировщик может оценить изменение и обнаружить, что оно дефектно. Если это так, он вносит свое имя и перемещает состояние обратно в «Сортировку». Эта обратная стрелка создает цикл, как показано на рис. 6.11.

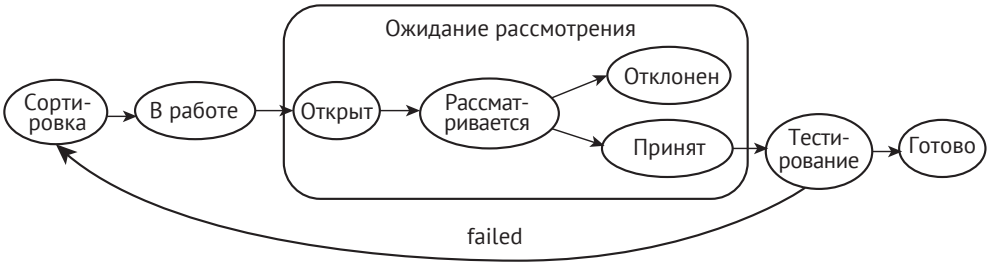


**Рис. 6.11.** Неудачное прохождение теста перемещает ошибку обратно в начало рабочего процесса

Этот цикл означает, что мы не можем написать тот же вид условной проверки, который видели ранее. Мы не можем с уверенностью сказать, что если ошибка находится в «Сортировке» или «В работе», поле «Тестирование» будет пусто. Оно будет пустым при первом прохождении, но не в том случае, если проверка ранее окончилась неудачей. Хуже того, что произойдет, если другой тестировщик проверит ошибку во второй раз? Какое имя окажется в поле?

## Сбор данных во время переходов

Условная проверка заставляет нас объявлять поля как пустые, когда в них на самом деле записываются необходимые данные. Как только состояние переходит за определенную точку, мы хотим убедиться, что эти данные были получены. Мы никогда не хотим допустить, чтобы номер отслеживания был пустым после того, как заказ был отправлен. Мы также не хотим, чтобы после того, как ошибка была протестирована, поле тестировщика было пустым. Чтобы изменить эти поля так, чтобы они не допускали пустых значений (null), нам нужно удалить их из их сущностей и поместить в новый объект. Этот объект будет создан только тогда, когда сущность перейдет в целевое состояние.

Например, когда заказ отгружается, мы можем создать объект «Транспортировка», как показано на рис. 6.12. Номер отслеживания не является ни полем заказа, ни полем товара. Вместо этого он является полем объекта «Транспортировка». Этот объект записывает, какие товары были отправлены, и собирает другие данные, созданные в это время, такие как номер отслеживания.

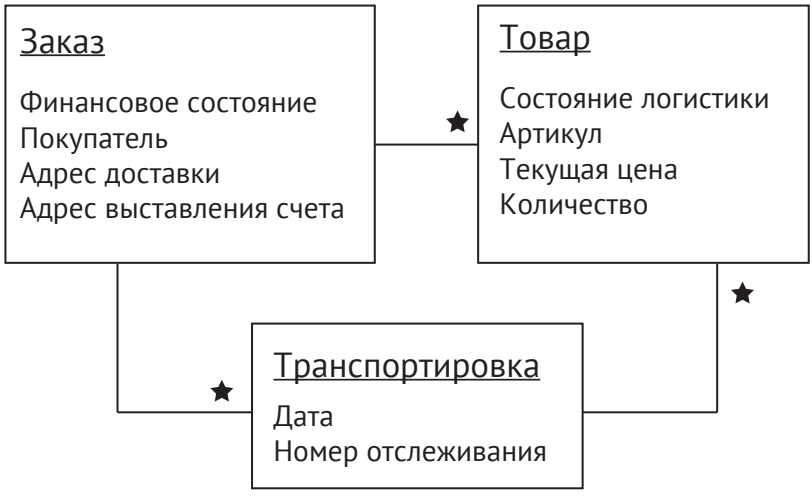


**Рис. 6.12.** Заказ имеет несколько транспортировок, каждая из которых имеет непустой номер отслеживания

С помощью этой модели мы можем убедиться, что номер отслеживания не является пустым. До того, как заказ отправляется, объекта «Транспортировка» не существует. Но после этого объект создается, и номер отслеживания должен быть заполнен.

Рассмотрим пример с системой отслеживания ошибок. В какой объект мы можем переместить поле «Тестировщик» (Tester)? Когда тестировщик не справляется с исправлением ошибки, он может создать объект «Неудача» (Fail). Этот объект может фиксировать не только тестировщика, но и описание

неудачи теста, ожидаемый результат и, возможно, скриншоты, журналы и другие вспомогательные материалы. И наоборот, когда тестировщик принимает исправление ошибки, он может создать объект «Пройден» (Pass). Этот объект фиксирует тестировщика и любые дополнительные примечания. Эти два объекта показаны на рис. 6.13.

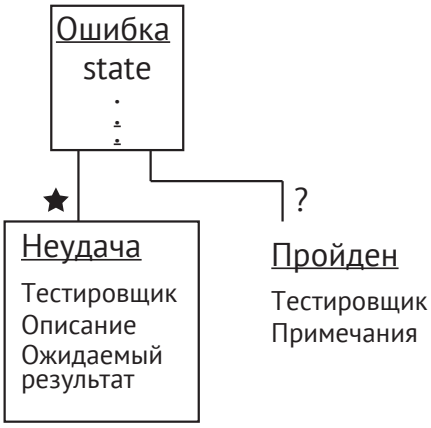


**Рис. 6.13.** Ошибка имеет много объектов Fail и ноль или один объект Pass. Эти объекты фиксируют тестировщика

Эта модель создает два новых объекта. Каждый объект имеет поле «Тестировщик», которое не может быть пустым (null). У нас больше нет условной проверки; до того, как ошибка будет протестирована, эти объекты даже не существуют. Кроме того, мы решили проблему с циклами состояний. Каждый раз, когда ошибка тестируется, мы создаем новый объект. У каждого из них может быть свой тестировщик, а предыдущие значения никогда не перезаписываются.

## Неизменяемые переходы состояний

Введя новые объекты, мы решили проблему условной недействительности. До того, как сущность перейдет за пределы определенного состояния, дополнительный объект не существует. После этого объект содержит непустые поля. Нам больше не нужно хранить поле внутри сущности. Это означает, что нам не нужно ухудшать объявление поля, чтобы разрешить более слабый тип.

Мы также решили проблему циклов. Каждый раз, когда сущность переходит через итерацию, она накапливает больше данных. Каждый переход записывается в отдельный объект. Предыдущие переходы не перезаписываются.

В любом случае новый объект является неизменяемым. Каждый из этих объектов представляет собой исторический факт. Нет причин возвращаться в прошлое и изменять эти факты. Мы заменили изменение созданием объекта. Возможно, теперь мы можем использовать эти неизменяемые объекты, чтобы устранить даже изменяемое поле, записывающее само состояние.

## Вопрос, стоящий за состоянием

Проблемы многих свойств, многих сущностей и условной проверки привели нас к тому, что мы записываем информацию о переходе состояния в неизменяемый объект. В настоящее время наши модели имеют одно или несколько изменяемых полей состояния для каждой сущности, в дополнение к неизменяемым историческим записям. Приложение изменяет состояние и добавляет исторические детали в одной транзакции, обеспечивая их согласованность. Это заставляет задаться вопросом – не является ли одно из этих действий избыточным?

## Перевод конечного автомата в историческую модель

Давайте рассмотрим каждый из примеров, которые мы изучали до сих пор, и построим историческую модель переходов состояний. Если мы можем вычислить состояние каждой сущности на основе исторических фактов, то можем быть уверены, что поле изменяемого состояния является избыточным. Имея эту уверенность, мы можем исключить избыточные поля из хранилища данных. Хранить то, что может быть вычислено из чего-то другого, – это значит допускать дефекты. Когда существуют избыточные поля, можно хранить несогласованный набор значений. Устранение лишнего поля устраняет этот класс дефектов.

Если у нас есть способ вычисления текущего состояния, мы можем задать более важный вопрос: что является причиной для определения состояния сущности в первую очередь? Исследуя, для чего на самом деле используется состояние, мы можем сопоставить эти вопросы с историческими фактами. Этот анализ покажет, что во многих областях нам вообще не нужно знать состояние сущности. Мы можем ответить на вопросы, связанные с состоянием, непосредственно на основе исторических фактов.

### Выполнение заказов

В системе выполнения заказов мы отслеживаем финансовое состояние заказа и логистическое состояние товара с помощью пары параллельных конечных автоматов. Давайте смоделируем каждый из этих автоматов как историю фактов, начиная с финансового состояния. Документы, влияющие на финансовое состояние, показаны на рис. 6.14. Я связал каждый документ с его предшественниками, чтобы представить причинно-следственные связи.

Финансовое состояние заказа можно записать как функцию от этих документов. Мы будем использовать экзистенциальные квантификаторы или утверждения вида «существует». Если существует счет-фактура для заказа, то заказ был оформлен. Если существует оплата за этот счет, то заказ был оплачен.

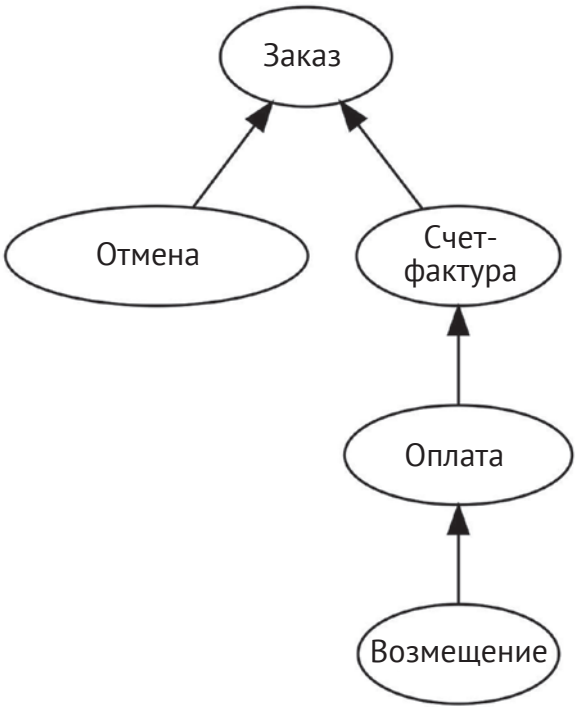


**Рис. 6.14.** Исторические факты, представляющие документы, которыми управляет бухгалтерский отдел

```
financial state (order) =
  if there exists Cancellation
    then Canceled
  else if there exists Invoice then
    if there exists Payment then
      if there exists Refund then
        Canceled
      else Paid
    else Invoiced
  else Ordered
```

Добавление нового документа в историю влияет на финансовое состояние заказа. Нам не нужны обработчики для получения документов и обновления состояния. Нет никакой машины для обработки сообщений. Состояние – это просто декларативная функция о существовании документов.

Состояние логистики немного сложнее. Мы записали номер отслеживания в неизменяемом объекте «Транспортировка», чтобы указать, что для данного заказа было отправлено несколько товаров. Диаграмма типов фактов на рис. 6.15 сохраняет эту связь между отправкой, заказом и отправленными товарами, а также учитывает другие действия, влияющие на состояние логистики.

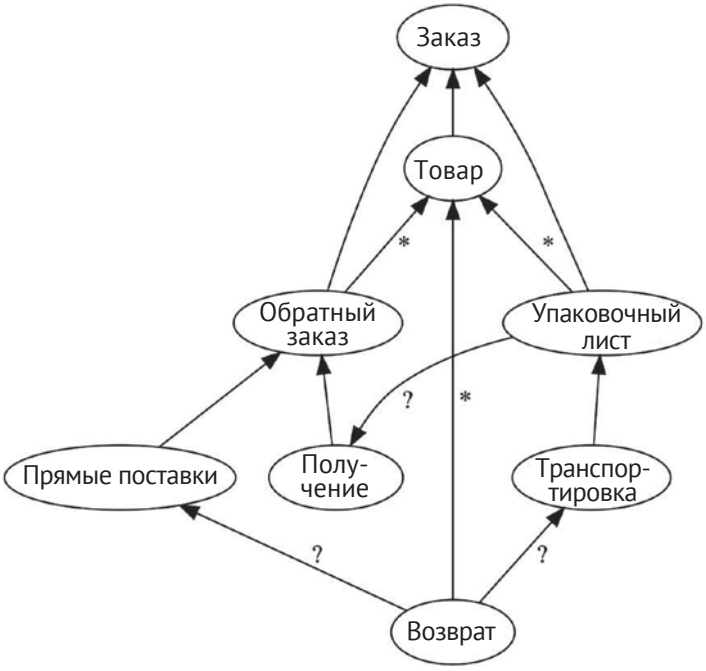


**Рис. 6.15.** Исторические факты, представляющие действия отдела доставки

Когда отдел доставки забирает товары со склада, он составляет упаковочный лист. Таким образом, наличие упаковочного листа указывает на то, что эти товары были отобраны. Позже этот комплект будет отгружен, и тогда он получит номер для отслеживания.

Товары, отсутствующие на складе, будут заказаны задним числом. Набор таких товаров может быть отгружен или получен. В случае получения склад создает новый упаковочный лист, в результате чего будет произведена транспортировка.

Товар может быть возвращен как при прямой поставке, так и при отгрузке со склада. Не обязательно возвращать всю партию, поэтому при возврате необходимо указать, какие товары были включены. История фактов рассказывает историю. Следующая функция определяет текущее состояние товара на основе существования этих фактов:

```
logistics state (item) =
  if there exists PackingSlip
    if there exists Shipment
      if there exists Return
        then Returned
      else Shipped
    else Picked
  else if there exists Backorder
    if there exists Receipt
      then Received
    else if there exists DropShipment
      if there exists Return
        then Returned
      else DropShipped
    else BackOrdered
  else Ordered
```

Эти две истории живут бок о бок в системе выполнения заказов. Бухгалтерский отдел занимается в основном финансовыми документами, в то время как отдел доставки занимается логистическими событиями. Они пересекаются только при сверке платежей с отгрузками и возмещений с возвратами. С помощью одного дополнительного запроса мы можем определить заказы, которые находятся вне баланса и должны быть исправлены. Мы напишем этот запрос немного позже. А пока давайте разработаем историческую модель для системы отслеживания изменений в программном обеспечении.

## Отслеживание изменений в программном обеспечении

Взглянув еще раз на систему отслеживания изменений в программном обеспечении, мы можем определить переходы состояний, которые происходят в отношении ошибки или коммита. Мы определили два таких перехода – Pass и Fail. Остальные переходы выглядят как стрелки на диаграмме переходов состояний. На рис. 6.16 мы обозначили эти стрелки, чтобы видеть весь набор.

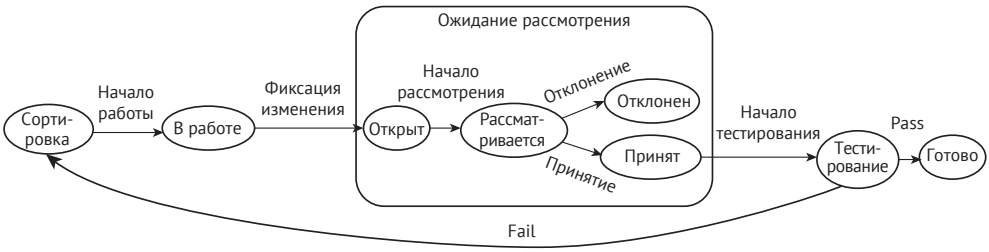


**Рис. 6.16.** Переходы в системе отслеживания изменений в программном обеспечении обозначаются действиями, которые их вызывают

Если перейти от диаграммы переходов состояний к графу типов фактов, то каждый из этих переходов становится фактом. Мы связываем факты с сущностями, на которые они влияют. Мы также связываем каждый факт с переходом состояния, который предшествовал ему. Это гарантирует, что мы не можем взять переход состояния слишком рано. Полученный граф типов фактов представлен на рис. 6.17.

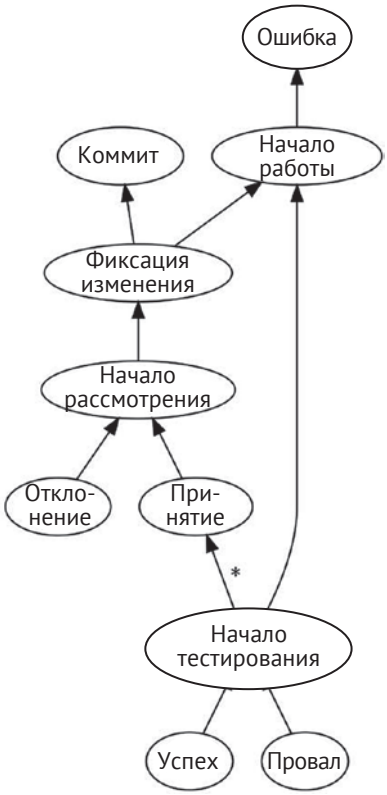


**Рис. 6.17.** Граф типов фактов, представляющий переходы состояний системы отслеживания изменений программного обеспечения

Переход «Начало тестирования» выполняется над ошибкой, и поэтому его предшественник является последним действием, выполненным над этой ошибкой – «Начало работы». Однако он зависит от того, что все коммиты, поставленные в результате этой работы, будут приняты. Поэтому у него также есть предшественник – коллекция фактов «Принятие». По этому графу исторических фактов мы можем определить состояние отдельного коммита или ошибки в целом. Давайте начнем с функции, которая выдает состояние коммита. Мы выражаем эту функцию в терминах факта «Фиксация изменения», так как нам не нужно рассматривать коммиты, которые не были отправлены.

```
state (pushCommit) =
  if exists BeginReview
       if exists Reject
              then Rejected
       else if exists Accept
              then Accepted
       else InReview
  else Open
```

Эта функция выражает тонкое, но важное структурное решение – отклонение запрещает принятие. Она проверяет факт «Отклонение» (Reject), прежде чем проверить факт «Принятие» (Accept). Если каким-то образом оба существуют, коммит отклоняется.

Теперь, когда у нас есть эта функция, мы можем использовать ее для выражения состояния ошибки. Состояние ошибки основывается не только на существовании действий, связанных с ошибкой, но и на состоянии всех коммитов. По этой причине функция сочетает в себе экзистенциальные и универсальные квантификаторы.

```
state (bug) =
  if there exists BeginWork
       if there exists PushCommit
              if for all pushCommit in branch, state(pushCommit) = Accepted
              and there exists BeginTest
                     if there exists Fail
                            then Triage
                     else if there exists Pass
                            then Done
                     else InTest
              else AwaitingReview
       else InProgress
  else Triage
```

И снова отклонение запрещает принятие. Более того, ошибка требует, чтобы все коммиты были приняты и чтобы мы явно начали тестирование. Если любая из проверок неудачна, ошибка остается в ожидании рассмотрения. Нам не нужно писать код для комбинации этих событий, чтобы продвигать состояние ошибки. Декларативная природа функции состояния позаботится об этом за нас.

## Причины для вычисления состояния

Это упражнение демонстрирует, что мы можем определить состояние сущности на основе существования исторических фактов. Учитывая это, мы должны исключить изменяемое поле состояния из записи сущности. Это исключает возможность того, что нам не удастся правильно обновить ее при вставке нового исторического факта.

Убрав поле изменяемого состояния, мы можем теперь рассмотреть, зачем оно нам вообще было нужно. При каких обстоятельствах мы будем вызывать эти новые декларативные функции состояния? Каков вопрос, для ответа на который мы использовали состояние?

## Обработка следующего действия

Модели, основанные на конечных автоматах, показывают нам одну из причин, по которой мы хотим знать состояние сущности, – чтобы понять, как реагировать на следующее действие. Обработчик сообщений обычно следует предсказуемой серии шагов:

- найти сущность (по идентификатору корреляции или какому-либо другому свойству сообщения);
- определить состояние сущности;
- проверить сообщение;
- выполнить операцию над сущностью;
- обновить состояние.

Состояние определяет, как выполнять последующие операции. Мы можем запустить декларативную функцию для определения текущего состояния, а затем на основе этого состояния определить стратегию обработки сообщения. Или же мы можем пропустить шаг и просто определить стратегию непосредственно из истории. Во многих случаях решение о том, как реагировать на действие, гораздо проще, чем определение состояния.

Например, в системе выполнения заказов реагирование на запрос на отмену заказа зависит от финансового состояния заказа. Если заказ оплачен, то мы выдаем возмещение. Если нет, то мы просто отменяем заказ. Определение состояния, как мы видели ранее, несколько сложнее. И все же определить, оплачен ли заказ, гораздо проще:

```
paid (order) =
  there exists Invoice
       such that there exists Payment
              such that there does not exist Refund
```

Более того, работа по определению того, оплачен ли заказ, необходима для поиска платежей, которые должны быть возмещены. Таким образом, мы можем свести все это к одному Factual-запросу:

```
query nonRefundedPayments(o: Order) {
  match p: Payment where p.invoice.order = o
       such that not exists r: Refund where r.payment = p
}
```

Когда мы получаем заявку на отмену заказа, то выполняем этот запрос. Если он находит какие-либо платежи, которые еще не были возмещены, мы выдаем возмещение за них. Это имеет побочный эффект перехода заказа в состояние «Отменен», если мы хотим запустить его. Но если мы можем определить, как обработать следующее действие, не запуская функцию состояния, то зачем ее запускать?

## Поиск рабочих элементов

Помимо определения того, как обрабатывать следующее действие для сущности, состояния часто используются для поиска всех сущностей, требующих следующего действия. Вместо того чтобы искать состояние для данной сущности, системы будут запрашивать все сущности, которые находятся в данном состоянии. Одна из распространенных причин для выполнения такого запроса заключается в том, чтобы представить результаты пользователю в виде списка рабочих элементов.

Например, система отслеживания изменений в программном обеспечении обычно отображает проблемы в виде «плавательных дорожек» (линеек состояний – *Прим. ред.*). Каждый столбец представляет собой состояние. Когда пользователь ищет работу, которую необходимо выполнить, он просматривает проблемы в данном состоянии и выбирает одну из них для продвижения вперед. А разработчик будет искать в «Сортировке» ошибки, которые готовы к обработке. Перебирать все ошибки и запускать функцию состояния было бы медленным способом предоставить этот пользовательский интерфейс. К счастью, запрос Factual дает нам эти результаты непосредственно из истории.

```
query bugsInTriage(s: Sprint) {
  match b: Bug where b.sprint = s
       such that not exists bw: BeginWork where bw.bug = b
}
```

Для этого запроса мы помещаем все ошибки в спринт[1]. Это дает нам отправную точку для запроса. Сначала мы ищем все ошибки в спринте, а затем ограничиваем их только теми, работа над которыми еще не началась. Другими словами, нам нужны только те рабочие элементы, которые примут наше следующее действие. Разработчик может выбрать любой из них и создать новый факт «Начать работу», тем самым удалив его из «плавательной дорожки».

Это был довольно простой пример. А как насчет чего-то более сложного? Например, тестировщик может искать все ошибки, которые находятся в стадии тестирования, чтобы потом выбрать одну для проверки. Ошибка находится в тестировании только в том случае, если ни один коммит не был рассмотрен и отклонен.

[1] Спринт (sprint) – это промежуток времени (месяц или меньше, например 2 недели), за который создается инкремент продукта. Инкремент продукта – это определенный ощутимый результат, к которому должна прийти команда в течение спринта. https://gb.ru/posts/gibkie-metody-razrabotki. – *Прим. перев.*

```
query bugsInTest(s: Sprint) {
  match bw: BeginWork where bw.bug.sprint = s
       such that not exists r: Reject
              where r.beginReview.pushCommit.beginWork = bw
  then bt: BeginTest where bt.beginWork = bw
       such that not exists p: Pass where p.beginTest = bt
       and not exists f: Fail where f.beginTest = bt
}
```

Последнее предложение фильтрует список на основе следующего действия. Нас интересуют только ошибки, которые не были пройдены или провалены. Запросы к рабочим элементам всегда включают условие «не существует», основанное на следующем действии. Как только мы выполним следующее действие, запрос больше не будет возвращать результат. Это обновляет пользовательский интерфейс и удаляет рабочий элемент из списка пользователя.

### Выполнение компенсирующих транзакций

Последняя причина, по которой необходимо знать состояние сущности, заключается в том, чтобы определить, есть ли какие-либо компенсирующие операции, которые должны быть применены. Это одна из функций, для которых был первоначально изобретен шаблон Saga[1]. Большинство систем управления базами данных предоставляют механизм для выполнения нескольких операций в одной атомарной транзакции. Эти транзакции предназначены для кратковременного выполнения. Сохранение транзакции открытой в течение длительного времени может блокировать выполнение других операций, что серьезно влияет на масштабируемость вашего решения. Шаблон Saga связывает компенсирующие транзакции с промежуточными шагами, чтобы их можно было откатить в случае возникновения проблемы. Компенсирующие транзакции позволяют фиксировать промежуточные транзакции базы данных и при этом обрабатывать длительные действия.

Другой способ подумать о компенсирующих транзакциях – рассмотреть, является состояние желательным или нет. Как правило, транзакции базы данных предотвращают систему от входа в нежелательные состояния, поскольку транзакцию можно откатить. В Saga, однако, транзакции базы данных выполняются чаще, чтобы улучшить масштабируемость. Возможно, что последовательность небольших транзакций базы данных приведет к тому, что система окажется в нежелательном состоянии. Компенсирующая транзакция – это корректирующее действие, которое может быть предпринято, если система окажется в таком состоянии.

Один пример из системы выполнения заказов связан с проверкой возвратов с возмещением. Нежелательно, чтобы финансовое состояние заказа было «Оплачено», в то время как логистическое состояние его товаров было «Возвращено». Мы можем прийти к такому состоянию двумя способами – получить оплату за товары, которые были возвращены, или получить возврат товаров,

[1] *Hector Garcia-Molina, Kenneth Salem*. Sagas. Department of Computer Science, Princeton University. 1987.

которые уже оплачены. Реализация Saga на основе конечного автомата будет искать каждую из этих ситуаций в отдельном обработчике. Обработчик оплаты будет искать товары в состоянии «Возвращено», а обработчик возвратов будет искать заказы в состоянии «Оплачено». Но при этом дублируется логика, которую можно было бы просто выразить в одном месте. Следующий запрос Factual определяет товары, которые как оплачены, так и возвращены:

```
query itemsRequiringRefunds(s: Seller) {
  match i: Item where i.order.seller = s
       such that there exists p: Payment where p.invoice.order = i.order
              such that not exists rf: Refund
                     where rf.payment = p and rf.items = i
       and there exists r: Return where r.item = i
}
```

Сервис выполняет этот запрос, чтобы определить, за какие товары нужно вернуть деньги. Если он возвращает какие-либо товары, то выполняется это возмещение. После этого добавление факта «Возмещение» удаляет товар из предыдущего запроса. Сервису не нужно было определять состояние товаров и их заказов, чтобы найти товары, требующие компенсирующих операций. Он выполняет этот запрос непосредственно в истории и затем действует в соответствии с результатами.

В большинстве случаев я обнаружил, что прямой запрос истории дает именно ту информацию, которая мне нужна, без определения состояния. Чтобы определить, должна ли кнопка быть включена, просто запросите, что ее действие еще не существует. Чтобы распределить очередь по рабочим узлам, сделайте запрос на еще не отработанные элементы. Исторические запросы являются более прямыми и менее ошибочными, чем управление конечными автоматами. И это не только по аналитическим причинам, они также имеют значительное техническое преимущество.

## Единый источник истины

Мы определили альтернативу решениям бизнес-проблем на основе конечных автоматов. Во-первых, мы смоделировали переходы не как изменения в изменчивом состоянии сущности, а как неизменяемые записи. Затем написали декларативные функции, которые запрашивают наличие и отсутствие таких записей. Эти функции отвечают на два вопроса – каково следующее действие для данной сущности и какие сущности могут принять данное действие. Найти ответы на эти вопросы непосредственно из истории оказалось проще, чем отслеживать состояния. В качестве дополнительного преимущества мы обнаружим, что это также решает некоторые технические проблемы, которые возникают в распределенных системах.

Чтобы обрабатывать запросы, конечный автомат должен делать две вещи. Во-первых, он должен точно знать состояние объекта. Во-вторых, он должен определить, корректен ли запрос, когда объект находится в этом состоянии. Как следствие клиенты имеют меньше автономии. Они должны полагаться

на привилегированный набор узлов для обработки запросов от своего имени. Клиенты должны обращаться к единому источнику истины, чтобы узнать, что произошло в результате их действий.

## Оркестраторы

Многие распределенные системы, основанные на конечных автоматах, используют архитектуры, управляемые сообщениями. В такой системе каждый шаг в процессе связан с *командным* сообщением. Узел, называемый *оркестратором*, получает каждое сообщение и выполняет соответствующую транзакцию. Оркестратор загружает текущее состояние из объекта, определяемого *идентификатором корреляции* в команде. Он определяет, какую транзакцию применить, основываясь на состоянии сущности. Транзакция изменяет сущность, а конечный автомат определяет следующее состояние.

### Согласованное состояние

Когда мы изучали теорему CAP в главе 4, то определили согласованность как свойство, при котором два узла в распределенной системе сообщают о том, что объект находится в одном и том же состоянии. Чтобы оркестратор знал состояние объекта, он должен достичь согласованности с другими оркестраторами в системе. Это условие не может быть смягчено. Конечной согласованности недостаточно, потому что конечный автомат должен обработать команду и предоставить окончательный результат. Чтобы добиться согласованности, оркестраторы часто используют общую базу данных.

Оркестратор должен иметь возможность получить блокировку против объекта, чтобы определить, как обработать команду. По этой причине нередко все команды отправляются в центральный набор оркестрантов. Этот набор оркестраторов и база данных, в которой они поддерживают состояние объекта, становятся единым источником истины. Это единственный авторитет в отношении состояния данных объектов.

### Центральная проверка

Конечный автомат определяет, какие операции допустимы в зависимости от состояния объекта. Оплата может быть применена, только если заказ находится в состоянии «Выставлен счет». Ошибка может быть провалена, если она находится в состоянии «Тестирование». В распределенной системе пользователи, инициирующие эти операции, не находятся вместе с конечными автоматами. Они используют клиентов.

Клиент отдает команды на обработку оркестратору. Поскольку оркестраторы должны быть согласованы друг с другом, теорема CAP говорит нам, что они должны стать недоступными в случае разделения сети. Чтобы преодолеть это, команды часто ставятся в очередь. Оркестратор обработает команду позже, как только разделение сети будет устранено.

Поскольку результат команды зависит от состояния объекта, клиент не может точно предсказать, что произойдет. Клиенты находятся вне единого источника истины. Они должны ждать, пока команда достигнет оркестратора, а за-

тем – пока результат вернется к клиенту. А к тому времени, когда результаты возвращаются, пользователь уже перейдет к другим действиям.

Полагаясь на согласованное состояние для подтверждения операций, решение на основе конечного автомата жертвует автономностью. Клиент не может предсказать результат запроса самостоятельно. Он не может определить, будет запрос успешным или неудачным, так как другие запросы, о которых он не знает, могут перевести объект в иное состояние. Вместо этого клиенты должны полагаться на единственный источник истины, подавать команды в очередь и ждать результатов.

## Сходящиеся истории

Вместо того чтобы полагаться на централизованную статическую модель как на единственный источник истины, мы можем позволить каждому клиенту представлять свою собственную истину. Истина – это все, что сделал пользователь, в обязанности оркестратора не входит проверка действий пользователя. Это ответственность системы – понять, что произошло, и объединить эти истории в целостную историю.

### Определение неизменяемых записей

Для начала мы моделируем каждое действие пользователя как неизменяемую запись. Эта запись фиксирует информацию, которая была получена в момент совершения действия. Каждая запись ссылается на предыдущие действия как предшественники. Это не просто список всех событий, которые произошли в прошлом; эти предыдущие действия – это именно те, которые привели к новому действию. Предшественники представляют собой информацию, которой располагал пользователь при принятии решения.

Например, факт «Отправка» содержит номер отслеживания. Эта информация создается, когда пользователь предпринимает действие. Кроме того, платеж ссылается на предшественника «Счет-фактура», но не связан ни с какими отгрузками, которые, возможно, уже произошли. Счет-фактура причинно связана с платежом, но отгрузка – нет.

### Запрос для следующего действия

Когда неизменяемые записи смоделированы, напишите запрос для каждого вида действия. Запрос для всех записей, которые являются предшественниками данного действия. Добавьте предложение, включающее только те, для которых действие не существует. Этот запрос скажет вам, является ли это действие следующим действием для объекта.

Например, для коммита запрос на «Начать рассмотрение» без последующего «Отклонить» или «Принять». Если этот запрос возвращает одну или несколько записей, значит, коммит все еще находится на рассмотрении. Следующим действием будет принятие или отклонение. Более того, записи из списка, возвращенного запросом, являются предшественниками записей «Принять» или «Отклонить», которые будут созданы.

Напишите другой запрос, который начинается выше по цепочке. Вместо того чтобы начать с коммита, начните со спринта. В этом запросе будут перечис-

лены все объекты, для которых требуется определенное следующее действие. Это обеспечит набор рабочих элементов для представления пользователю.

### Локальное фиксирование действий

Когда пользователь совершает действие, немедленно добавьте запись в локальную историю. Создание записи будет влиять на результаты запросов. Пользователь сразу же увидит, что объект удален из одного набора рабочих элементов и добавлен в следующий. Он заметит, что новое следующее действие объекта продвинулось вперед. Это дает ему немедленную обратную связь о том, что его действия выполнены.

Пользователь принимает решение на основе имеющейся у него информации. Система не должна делать паузу, чтобы проверить текущее состояние или принять удаленную блокировку, дабы убедиться, что действия пользователя согласованы. Она не должна помещать запись в очередь и ждать, пока оркестратор обработает ее позже. Зафиксируйте действие в истории на клиенте, и пусть параллельные истории сходятся.

### Определите компенсирующие действия

Наконец, определите нежелательные состояния, которые могут возникнуть из-за сходящихся историй. Такие состояния могут возникнуть из-за того, что мы предоставили клиентам самостоятельность в действиях на основе информации, которой они располагают на данный момент. Напишите запрос, чтобы найти объекты в нежелательных состояниях, а затем разработайте процессы для выполнения компенсирующих операций.

Возможно, что у заказа есть и возмещение, и невозвращенная отправка. Создайте запрос, ищущий заказы в этом состоянии. Представьте его результаты специалисту, побуждая его позвонить клиенту и попросить его либо вернуть товар, либо оплатить другой счет. Аналитик должен сам определить соответствующие компенсации для таких ситуаций, а не ответственность конечного автомата за попытку их предотвратить.

Пользователи являются источником истины. Они распределены по всей системе. Они не участвуют в деятельности конечных автоматов. Их не волнует текущее состояние объекта. Им просто нужно знать, что произошло и как интерпретировать эти параллельные истории. Им нужна уверенность в том, что их действия будут признаны и станут частью этой сходящейся истории.

# Глава 7

# Безопасность

Общим подходом к обеспечению безопасности приложений является управление доступом на основе ролей (role-based access control – RBAC). В этой системе администратор назначает людей на роли, а затем разрешает этим ролям выполнение определенных действий в системе. По мере внедрения неизменяемых архитектур RBAC становится все более сложной задачей. Требование к администратору назначать роли и давать разрешения уменьшает автономию индивидуальных пользователей. Обращение к единому источнику истины для этих ролей и разрешений снижает автономность клиентских узлов. Модель управления доступом начинает работать против преимуществ, за которые мы так упорно боролись.

RBAC обычно применяется на уровне организации. Группа администраторов определяет роли и операции в организации. Они управляют набором ресурсов, на которых выполняются операции. Эта организация является единственным выгодоприобретателем системы. В многопользовательской среде, однако, разделение ролей и обязанностей становится гораздо сложнее. Многочисленные арендаторы могут не договориться о едином органе управления доступом к своим ресурсам. Вместо этого они будут стремиться сохранить автономию своих собственных активов. Это приведет их к отказу от централизованной формы управления доступом и к более распределенным моделям доверия.

Наше стремление к автономии заставляет нас использовать децентрализованную модель контроля доступа. Расширение распределенных систем на нескольких арендаторов устраняет организационные структуры, на которые мы могли бы положиться в противном случае. Поэтому мы ищем решение. Вместо модели доступа на основе ролей мы находим вдохновение в инфраструктуре открытых ключей (public key infrastructure – PKI) и делегировании полномочий. С помощью этих инструментов мы можем построить систему безопасности на основе модели неизменяемых фактов.

## Доказательство авторства

Историческая запись представляет собой решение, принятое человеком в распределенной системе. Прежде чем мы сможем определить, стоит ли доверять этому конкретному решению, мы должны иметь некоторую уверенность

в личности человека, который его принял. Мы ищем доказательства авторства факта. Современные цифровые системы полагаются на *инфраструктуру открытых ключей* (PKI) для обеспечения такого доказательства. PKI основана на существовании функции-ловушки – математической функции, которую легко вычислить в одном направлении, но трудно инвертировать.

## Ключевые пары

Предположим, что у меня есть пара функций. Каждая функция является обратной по отношению к другой, если подстановка результата одной функции в другую дает исходное значение. Например, функции `x+3` и `x-3` представляют собой такую пару. Прибавление трех, а затем вычитание трех возвращает исходное значение. Вы можете придумать еще много примеров и, вероятно, придумать несколько различных способов создания новых пар.

Если я дам вам одну из этих функций, скажем `x+3`, вы, вероятно, сможете сказать, что такое ее обратная функция. Вычислить обратную такой простой функции несложно.

Но что, если я дам вам функцию типа $x^{37} \bmod 1829$? Вам может потребоваться некоторое время, чтобы понять, что такое ее обратная функция (инверсия). Компьютер с правильным алгоритмом сможет найти ее достаточно быстро – $x^{823} \bmod 1829$, как показано на рис. 7.1. Но если я сделаю эти числа значительно больше, то даже самому мощному цифровому компьютеру будет очень трудно найти инверсию.

$$36 \rightarrow x^{37} \bmod 1829 \rightarrow 842 \rightarrow x^{823} \bmod 1829 \rightarrow 36$$

$$42 \rightarrow x^{823} \bmod 1829 \rightarrow 858 \rightarrow x^{37} \bmod 1829 \rightarrow 42$$

**Рис. 7.1.** Некоторые функции, являющиеся инверсиями друг друга, могут быть использованы в асимметричной криптографии

Функции такой формы являются примерами функций-ловушек. Они были созданы с помощью протокола, известного как RSA, названного в честь его изобретателей Ривеста (Rivest), Шамира (Shamir) и Адлемана (Adleman)[1]. Вам действительно легко вычислить модульную экспоненту. Очень трудно вычислить ее инверсию (известную как дискретный логарифм). Только потому, что я создал эту пару в одно и то же время, я смог сам найти инверсию.

Если вы внимательно посмотрите на эти функции, то увидите, что у них есть верхняя граница. Функция $x^{37} \bmod 1829$ имеет обратную, только если $x < 1829$. Это следствие *принципа «голубятни»*. Модуль 1829 ограничивает количество возможных результатов, которые может вернуть функция, он определяет, сколько голубиных нор у вас есть. Если вы попытаетесь поместить

[1] *R. L. Rivest, A. Shamir, and L. Adleman.* A Method for Obtaining Digital Signatures and PublicKey Cryptosystems. Communications of the ACM. February 1978.

больше голубей в функцию, то в одной голубятне окажутся по крайней мере двое. Эта функция намеренно построена так, чтобы каждый вход имел явный выход, что необходимо для того, чтобы убедиться, что она имеет инверсию в той же области.

RSA дает нам протокол для одновременной генерации функции и ее инверсии. Если я дам вам одну из функций, вам будет трудно найти ее инверсию, это функция-ловушка. Мы будем называть ту функцию, которую я передаю, *открытым ключом*. Функция, которую я храню у себя, является *закрытым ключом*. Я могу использовать эту пару обратных функций для доказательства авторства.

## Дайджест

Если выполнить хеш-функцию над потоком данных, то получится дайджест. Дайджест будет иметь фиксированный размер, определяемый выбранной вами хеш-функцией. Важно выбрать хеш намного меньше, чем предел пары открытый/закрытый ключ. Дополните этот дайджест некоторыми случайными данными и введите их в функцию закрытого ключа. Результат – это подпись.

Если кто-то знает ваш открытый ключ, он может проверить вашу подпись. Все, что для этого необходимо, – это прогнать подпись через функцию открытого ключа и получить обратно свой дайджест, как показано на рис. 7.2. Затем он может вычислить дайджест и посмотреть, совпадает ли он с вашим. Если да, то у него есть уверенность в том, что сообщение пришло от вас. Это работает, потому что было бы очень трудно инвертировать ваш открытый ключ, чтобы найти ваш закрытый ключ. Без вашего закрытого ключа он не сможет создать подпись, содержащую корректный дайджест.


**Рис. 7.2.** Прогон дайджеста документа через закрытый ключ позволяет получить цифровую подпись

Если вы хотите, чтобы потребители неизменяемой записи знали, что она получена от вас, вы можете создать дайджест. Процедура должна быть повторяемой, чтобы гарантировать, что все создавали один и тот же дайджест. Поэтому мы должны выбрать каноническую форму для неизменяемых записей. Одна из процедур, которую мне нравится использовать, выглядит следующим образом:

- сериализуйте запись в JSON;
- включите тип записи в качестве поля;

- замените предшественников объектами, имеющими только поле ref, значением которого является дайджест предшественника;
- отсортируйте коллекции предшественников по их дайджесту и удалите дубликаты;
- отсортируйте поля по алфавиту;
- удалите лишние пробельные символы;
- закодируйте текст с помощью UTF-8;
- вычислите хеш SHA-256 потока.

Например, сообщение на форуме может быть сериализовано следующим образом до удаления пробелов:

```
{
  «author»: {
       «ref»:  «MSHFIB0X5Jkup0Yu7ZZuIKJHVtow3vtAK/7f4GYmKVqdcKMcVg9AURmgU9
RQA
       tJwQjaYguJSJZzlwFctOTqrCw==«
  },
  «forum»: {
       «ref»:   «PQu2HVVqExA0r1kO9lK+rWHzui5Ysd07+g5VkgnNRsJqPnpsy5rzjSfIp
nd79
       aea8jjoPe+YIiouOz3xcJvQUQ==«
  },
  «text»: «Posted my first forum message.»,
  «type»: «ForumPost»
}
```

После удаления пробельных символов, применения кодировки UTF-8 и получения хеша SHA-256 вы можете создать подпись, используя свой закрытый ключ. Любой потребитель этой записи вычислит тот же самый дайджест. Если бы у него был ваш открытый ключ, он мог бы проверить вашу подпись. Поскольку только человек с вашим закрытым ключом может создать действительную подпись, потребитель может быть уверенным, что сообщение пришло от вас.

Сама неизменяемая запись не содержит подписи. И не может, потому что подпись создается из дайджеста содержимого записи. Вместо этого подпись хранится в оболочке. Когда один узел передает коллекцию неизменяемых записей другому, он предоставляет также список открытых ключей и подписей.

## Авторизация

Доказав, что вы являетесь автором факта, вы можете попытаться доказать, что вам разрешено это делать. Как вы можете подтвердить свое право на авторизацию в распределенной системе? Если бы мы могли обратиться к какому-то центральному органу, то, возможно, он мог бы подтвердить ваше требование. Но чтобы сохранить автономию, мы хотели бы избежать такого органа. Поэтому мы должны создать средство авторизации, которое получатель может проверить самостоятельно.

Для начала мы определим средство обмена открытыми ключами. Когда узел взаимодействует с агентом, он должен иметь возможность проверить утверждения об авторстве. Затем получатель обратится к набору правил, чтобы определить, какие агенты уполномочены производить какие записи. Эти правила полностью основаны на связанных записях.

## Факты принципала

Самый простой способ обмена открытыми ключами – сделать их частью исторической модели как таковой. Каждый агент, способный производить факты, является принципалом. Пользователи являются принципалами, автономные службы являются принципалами, уполномоченные объекты являются принципалами. Каждый принципал создает исторический факт, который представляет его самого.

Факт принципала содержит в качестве поля открытый ключ пользователя. Обычно он не содержит никаких дополнительных полей. Поскольку идентификация неизменяемой записи вытекает из ее содержимого, идентификацией принципала является его открытый ключ. Например, пользователь форума будет представлен фактом, показанным на рис. 7.3.

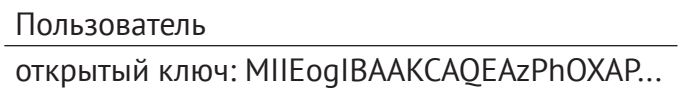


**Рис. 7.3.** У факта принципала есть открытый ключ

Факт принципала не нуждается в подписи. Подпись не доказывает ничего полезного. Факт принципала не содержит никаких дополнительных утверждений, поэтому подписант лишь подтверждает, что это его открытый ключ.

## Запрос авторизации

Когда узел получает факт, новый факт считается *оспариваемым*. Разрешение автора на создание этого факта остается под вопросом. Узел должен проверить, что автор имеет полномочия на создание оспариваемого факта. Для этого он выполнит запрос авторизации. Этот запрос определяет, какие факты принципалов уполномочены создавать оспариваемый факт. Если оболочка данного факта содержит подпись от уполномоченного принципала, то факт является *разрешенным*. Только разрешенные факты хранятся, используются в последующих запросах и пересылаются другим узлам.

Например, сообщение на форуме, которое мы недавно наблюдали, является примером следующего типа факта:

```
fact ForumPost {
  forum: Forum
  author: User
  text: string
}
```

Чтобы убедиться, что сообщение подписано автором, каждый получатель будет выполнять следующее правило авторизации:

```
authorize p: ForumPost {
  match u: User where p.author = u
}
```

Запрос авторизации, наложенный поверх графа фактов, показан на рис. 7.4.

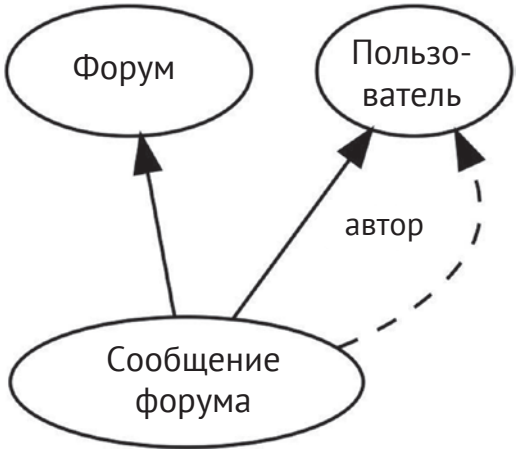


**Рис. 7.4.** Сообщение форума с запросом авторизации, который определяет, какой пользователь может его создать

Это правило предписывает получателю найти факт принципала, который является автором сообщения. Оно использует открытый ключ данного факта. Если оболочка содержит достоверную подпись к этому открытому ключу, то сообщение авторизовано. Если нет, то получатель немедленно отвергает факт.

При использовании правил авторизации, описанных ранее, можно предотвратить подделку сообщений другими лицами. Если ваш открытый ключ становится известным, то потенциальный фальсификатор может попытаться создать новое сообщение на форуме с вашим принципалом в качестве автора-предшественника. Однако, не обладая вашим закрытым ключом, он не сможет создать соответствующую подпись. Фальсификатору придется довольствоваться созданием новой пары из открытого и закрытого ключей. Такие сообщения будут авторизованы, но их автором будет другой принципал. Ни один получатель не будет обманут, полагая, что сообщение на форум было отправлено вами.

## Первоначальная авторизация

Хотя предыдущее правило авторизации предотвращает подделку, оно все равно позволяет любому пользователю размещать сообщения на форуме. В других случаях желательно ограничиться только ограниченным набором принципа-

лов. Не полагаясь на централизованный орган для управления этими ограничениями, мы обратимся к самой модели начальной авторизации.

Предположим, что вместо открытого форума мы хотим смоделировать персональный блог. Только создателю блога разрешено публиковать сообщения. Блог можно описать с помощью следующего факта:

```
fact Blog {
  creator: User
  identifier: guid
}
```

Правило авторизации проверяет, что автором блога действительно является его создатель:

```
authorize b: Blog {
  match u: User where b.creator = u
}
```

Теперь мы можем определить запись в блоге с помощью следующего факта:

```
fact BlogPost {
  blog: Blog
  author: User
  postedAt: datetime
}
```

Для этой области мы оставляем изменяемые свойства, такие как заголовок, текст и теги, в качестве отдельных фактов. Только дата и время отличают одну запись в блоге от других, написанных тем же автором в том же блоге.

Но самое главное, что теперь мы можем написать правило авторизации, которое позволяет публиковать записи в блоге только его создателю.

```
authorize p: BlogPost {
  match u: User
     where p.author = u
     and p.blog.creator = u
}
```

Правило авторизации для записи в блоге проходит по графу фактов, как показано на рис. 7.5.

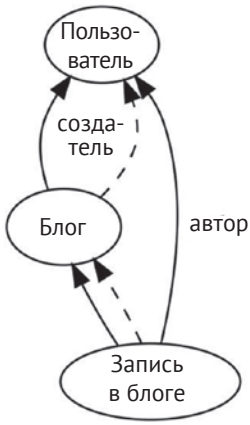


**Рис. 7.5.** Автором записи в блоге может быть только создатель блога

Этот запрос вернет автора только в том случае, если он также является создателем блога. Если кто-то попытается создать факт, утверждающий, что он принадлежит автору, не являющемуся создателем блога, то этот запрос не даст никаких результатов. Видя, что нет принципалов, уполномоченных подписать факт, получатель сразу же отклонит его. Создание блога обеспечивает первоначальный список авторизации без необходимости для какой-либо стороны консультироваться с централизованным администратором.

## Предоставление полномочий

Запросы авторизации, которые мы написали до сих пор, возвращают только создателей корневого объекта. Это идеальная отправная точка для децентрализованной системы; если вы ее создали, то только вы контролируете ее. Это полностью устраняет необходимость в администраторах для определения ролей, разрешений или для предоставления доступа к определенным ресурсам.

Но это только отправная точка. Для большинства областей авторизация не может оставаться лишь у создателя объекта. Создатель должен иметь возможность передать полномочия другой стороне. Такая передача может быть ограниченной или бессрочной. Если передача ограничена, то она ограничивается одним-единственным случаем. Но если она бессрочная, то полномочия сохраняются.

## Ограниченные полномочия

Создатель может предоставить другому принципалу полномочия для одного экземпляра объекта. Для этого создатель идентифицирует уполномоченного принципала как предшественника нового объекта. Правило авторизации предоставляет этому принципалу разрешение на создание преемников.

Лучше всего это видно на примере. Предположим, что создатель блога хотел бы пригласить гостя опубликовать пост на сайте. Он создает гостевой пост для выбранного пользователя.

```
fact GuestPost {
  blog: Blog
  guest: User
  createdAt: datetime
}
```

Только создатель блога может добавить гостевую запись. Клиенты используют следующее правило авторизации для обеспечения этого ограничения:

```
authorize gp: GuestPost {
  match u: User where gp.blog.creator = u
}
```

После этого гость получает право устанавливать заголовок, писать текст и выполнять другие последующие операции. Например, чтобы установить заголовок, гость создает факт следующего вида:

```
fact GuestPostTitle {
  post: GuestPost
  title: string
  prior: GuestPostTitle[]
}
```

Следующее правило уполномочивает гостя назначать этих преемников:

```
authorize t: GuestPostTitle {
  match u: User where t.post.guest = u
}
```

Диаграмма правила авторизации показана на рис. 7.6.

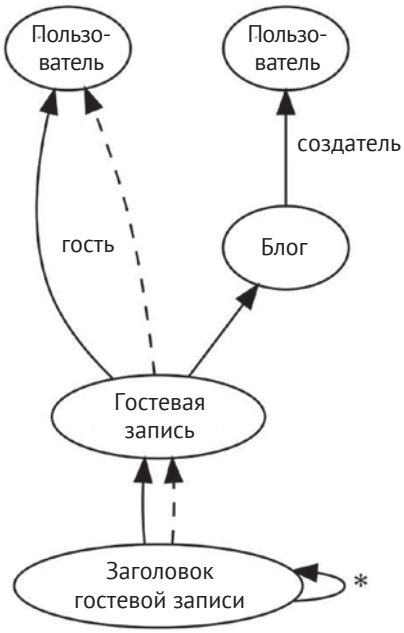


**Рис. 7.6.** Только гость может задать заголовок гостевого поста

Гость не имеет прав на создание дополнительных записей. Разрешение ограничено единственной дочерней записью, которая была создана от его имени.

## Неограниченные полномочия

Когда создатель объекта хочет разделить неограниченно полномочия с другими, он может создать факт, документирующий это решение. Разрешение должно быть подписано первоначальным создателем. В нем указывается объект, для которого предоставляются полномочия, и называется принципал, с которым будут разделены полномочия. Затем получателям должны быть даны инструкции по соблюдению дополнительных правил авторизации.

Предположим, например, что создатель блога хочет пригласить других писать в нем. Он не просто хочет дать им фиксированное количество гостевых записей. Он хочет разделить полномочия неограниченно. Для этого создатель блога выдает разрешение (грант).

```
fact BlogGrant {
  blog: Blog
  subject: User
  createdAt: datetime
}
```

Только создатель блога может выдать такой грант. Чтобы обеспечить это, мы пишем правило авторизации, требующее, чтобы грант исходил от создателя блога:

```
authorize bg: BlogGrant {
  match u: User where bg.blog.creator = u
}
```

Как только грант выдан, субъект должен быть авторизован для публикации. Мы пишем правило, утверждающее это.

```
authorize p: BlogPost {
  match bg: BlogGrant
       where bg.blog = p.blog
  then u: User
       where p.author = u
       and bg.subject = u
}
```

Это правило авторизации проходит зигзагами через граф фактов, как показано на рис. 7.7.

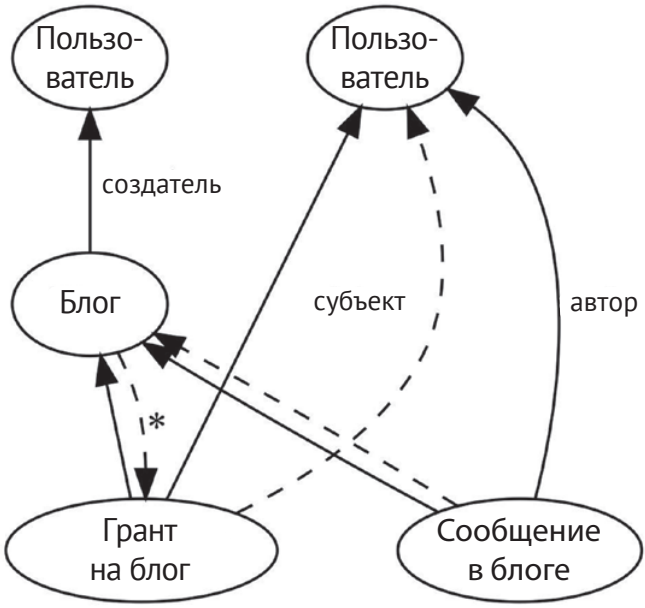


**Рис. 7.7.** Каждый субъект гранта имеет право на создание новой записи в блоге

Важно понимать, что это правило авторизации является *дополнением* к предыдущему правилу авторизации для сообщения в блоге. Клиент, соблюдающий оба этих правила, позволит создателю размещать сообщения и разрешит всем субъектам, наделенным полномочиями, размещать сообщения. У клиента не может быть только одного правила авторизации для каждого типа. Вместо этого объединение всех правил авторизации определяет набор принципалов, которые будут проверены. Если оболочка содержит подпись от любого члена этого набора, то факт авторизован.

## Транзитивная авторизация

Правило авторизации, упомянутое ранее, разрешает субъекту, имеющему грант на блог, размещать сообщения в чужом блоге. Однако это не ставит его в равные условия с создателем блога. Оно не позволяет ему затем распространять эти полномочия на других, выдавая дополнительные гранты. Если это будет желательной особенностью данной области, то клиентам нужно будет дать еще одно правило.

```
authorize next: BlogGrant {
  match bg: BlogGrant
       where bg.blog = next.blog
  then u: User
       where bg.subject = u
}
```

Как и все правила авторизации, это правило объединяется с другими для одного и того же оспариваемого факта. Другое правило уполномочивает создателя блога выдавать гранты. Это правило добавляет к нему – посредством объединения наборов – полномочия для субъектов распространять эти гранты на других. Рекурсивный характер этого определения подразумевает, что гранты могут быть распространены на любое количество поколений.

## Отмена

Все разрешения и гранты, задокументированные ранее, включают дату создания. Это не просто деталь аудита. Это проектное решение, позволяющее отменить разрешение. Одноразовое разрешение или бессрочный грант могут быть отменены последующим фактом. Правила авторизации просто должны включать в себя положение о том, что не существует, чтобы обеспечить это.

Если создатель блога хочет отменить предыдущее разрешение, то он может создать факт следующей формы:

```
fact BlogGrantRevoke {
  grant: BlogGrant
}
```

Создатель имеет право выдавать эти отмены для своего собственного блога. Правило авторизации, обеспечивающее это, выглядит следующим образом:

```
authorize r: BlogGrantRevoke {
  match u: User where r.grant.blog.creator = u
}
```

Если другие грантополучатели имеют аналогичные полномочия, то добавляется следующее правило:

```
authorize r: BlogGrantRevoke {
  match bg: BlogGrant
        where bg.blog = r.grant.blog
  then u: User
        where bg.subject = u
}
```

Теперь мы можем добавить в правило предложение `such that not exists`, чтобы этот факт отменял предыдущую авторизацию.

```
authorize p: BlogPost {
  match bg: BlogGrant where bg.blog = p.blog
        such that not exists r: BlogGrantRevoke where r.grant = bg
  then u: User
        where p.author = u
        and bg.subject = u
}
```

Добавление предложения `such that not exists` к правилу авторизации приводит к диаграмме, показанной на рис. 7.8.

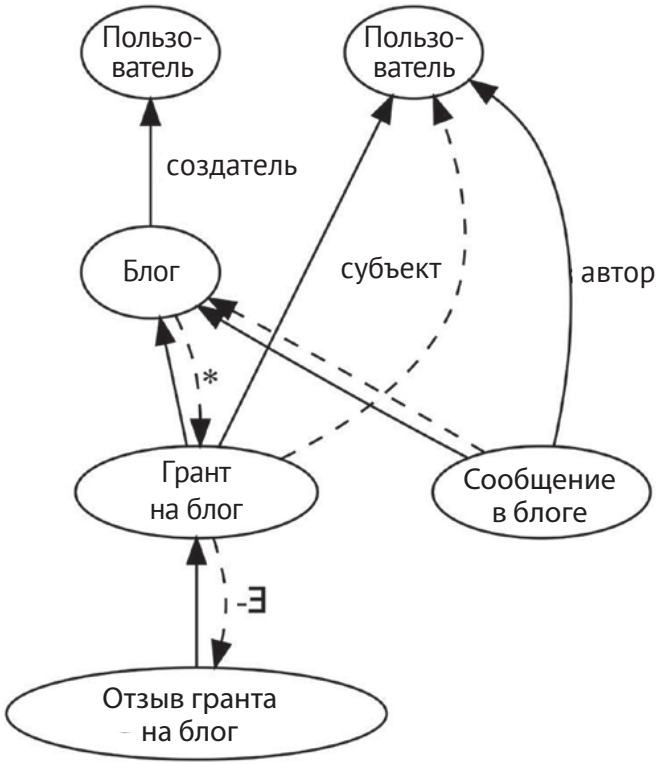


**Рис. 7.8.** Правило авторизации проверяет наличие отмены гранта на ведение блога

И именно здесь становится важным добавить к гранту такое отличительное поле, как «Время создания» (`createdAt`). Без него авторизация может быть предоставлена субъекту только один раз. После отмены полномочий не было бы возможности создать новый грант. Он был бы неотличим от аннулированного гранта.

## Авторизация при получении

Правила авторизации оцениваются сразу после получения факта. Они не проверяются позже, чтобы определить задним числом, должен ли был факт быть авторизован. Для этого есть две важные причины – сохранение и производительность.

Если бы правила авторизации оценивались задним числом, то отмена доступа привела бы к аннулированию всех предыдущих действий, которые были выполнены субъектом. В большинстве случаев это не является желаемым результатом. Сотрудник, которого увольняют, должен лишиться доступа. Однако вся работа, проделанная им для компании до этого момента, должна быть сохранена. Если бы правила авторизации были запущены для этих объектов после увольнения сотрудника, то эти факты стали бы недействительными.

Другим соображением является производительность. Ретроактивная оценка – это рекурсивная операция, отнимающая много времени. Если бы мы хотели использовать правила авторизации для проверки фактов каждый раз, когда они используются, нам пришлось бы применять их ко всем фактам, затронутым в запросе. Но чтобы подтвердить факт, мы должны запустить его правила авторизации. Эти правила сами являются запросами. Отслеживание этих запросов до фактов, которых они касаются, их правил авторизации и запросов к этим правилам приводит к взрывному росту количества проверок по историческому графу. Этот процесс не имеет гарантированного времени завершения, и даже когда он завершится, это отнимает невероятно много времени.

Хотя оценка при получении затрагивает эти две важные проблемы, она также вызывает проблему в отношении конечной согласованности. Один узел может определить, что факт действителен, в то время как другой решает, что он недействителен. Это происходит, когда один получил факт отмены, а другой – нет. Нет причинно-следственной связи между отменой и оспариваемым фактом. Ни один из них не является предшественником другого, поэтому любой из них может произойти первым. Как только это произойдет, никакие дальнейшие сообщения не приведут эти два узла к согласованности.

Учитывая опасность несогласованности, отмену следует использовать с особой осторожностью. Предоставьте такую возможность в моделях, которые вы разрабатываете, но предупредите пользователей, чтобы они использовали эту возможность очень осторожно. Встройте какой-нибудь другой механизм для отмены привилегий в системе. Например, контролируйте доступ сотрудников к закрытому ключу. Когда они увольняются, просто уничтожьте закрытый ключ, а не отменяйте права доступа. Или встройте в модель часы. Периодически обновляйте гранты. Если принципал покидает группу, просто пренебрегайте обновлением его гранта, а не отзывайте прошлые гранты.

# Конфиденциальность

Соображения безопасности, о которых мы говорили до сих пор, касались возможности записи данных в систему. Теперь мы обратим внимание на возможность чтения данных из системы. Как и в случае с записью, мы хотим контролировать чтение, не подчиняясь централизованной группе администраторов. Мы будем вести бизнес по развивающейся топологии сети, которая может включать узлы-партнеры, находящиеся вне нашего прямого контроля. И поэтому мы снова обращаемся к инфраструктуре открытых ключей за вдохновением для реализации доверия без центрального доверенного органа.

Стремление к автономии не является односторонним. С одной стороны, мы хотим иметь личный контроль над собой и своими устройствами. Мы хотим быть уверены, что можем действовать без необходимости подключаться к центральной системе учета. При работе с партнерами в распределенной системе мы должны ожидать, что они захотят такой же автономии для себя. Мы предоставили им такую автономию, применив политику «*доверяй, но проверяй*» – мы проверяли свои записи на соответствие согласованному набору правил авторизации. Но предоставление им автономии также дает им определенную степень доступа к нашей конфиденциальной информации. Теперь мы должны подумать о том, как сохранить конфиденциальность сообщений в такой среде.

## Недоверенные узлы

Конфиденциальность, как я ее здесь использую, означает наличие разумной уверенности в том, что информация, которую вы хотите передать определенной стороне, не будет перехвачена другими. Проблема в распределенной системе заключается в том, что предполагаемый получатель может не иметь прямой связи. Сообщения могут храниться на стороннем узле, который имеет большее время работы и доступность, чем устройства любой из сторон. Почта хранится на почтовых серверах, а не передается непосредственно от отправителя к получателю. Прямые сообщения публикуются в общих каналах. Записи, предназначенные для одной стороны, могут быть найдены на ненадежных посредниках.

Если мы предположим, что можем неявно доверять нашим сторонним провайдерам, то мы можем удовлетвориться простым шифрованием данных при передаче. Мы можем использовать безопасный протокол типа TLS для загрузки личного сообщения провайдеру социальных сетей, прекрасно зная, что оно будет расшифровано на сервере. Оно может даже храниться в открытом виде, чтобы быть повторно зашифрованным, когда получатель инициирует свое собственное TLS-соединение с тем же сервером. Но если мы не можем доверять нашим посредникам, то шифрования на транспортном уровне недостаточно.

## Асимметричное шифрование

Как мы видели в **доказательстве авторства**, открытый и закрытый ключи – это не что иное, как обратные функции. Подставьте число в одну из них, и она выдаст ответ, подставьте этот ответ в другую, и вы получите исходное число. Мы доказали авторство, применив сначала закрытый ключ. Мы можем отправлять личные сообщения, инвертируя процесс.

Если мы хотим отправить число получателю, то можем пропустить его через его открытый ключ. Результат может храниться на недоверенном узле без особых опасений. Любая сторона, не владеющая закрытым ключом, с большим трудом сможет инвертировать функцию, чтобы найти исходное значение. Это основа для достижения конфиденциальности в распределенной системе со сторонними узлами. Но прежде чем мы сможем применить этот протокол, нам придется столкнуться с проблемой ограничения размера.

### Асимметричное ограничение размера

Если вы помните, функции, которые мы создавали, используя протокол RSA, имели инверсии только в определенном диапазоне чисел. Функция x37 mod 1829, которую мы использовали в качестве открытого ключа, может производить инверсии только в области 0–1828. Принцип «голубятни» не позволял нам принимать любые входные данные, которые могли бы привести к двойному выходу. Поэтому этот открытый ключ может использоваться только с 1829 различными входами, он поддерживает лишь 10-битные сообщения. Реальных пар ключей RSA намного больше – 2048 или 4096 бит при обычном использовании. Тем не менее существует ограничение по размеру. Именно поэтому мы подписали хеш, а не оригинальное сообщение. И по той же причине мы не можем зашифровать оригинальное сообщение с помощью открытого ключа получателя.

Вместо этого мы применим к сообщению симметричное шифрование. Симметричные ключи не имеют такого ограничения на размер, как у пар асимметричных ключей. Они могут быть использованы для шифрования сообщения произвольной длины. Однако один и тот же ключ используется как для шифрования, так и для расшифровки сообщения. Поэтому мы должны хранить симметричный ключ в тайне. Это то, что мы передаем через открытый ключ получателя.

### Шифрование симметричного ключа

Итак, стратегия, которую мы используем для обеспечения конфиденциальности в распределенной системе, – это шифрование содержимого неизменяемых записей перед передачей. Сначала отправитель генерирует случайное число, которое будет использоваться в качестве симметричного ключа. Эффективные симметричные ключи могут быть значительно меньше, чем предельный размер такого же эффективного открытого ключа. Поэтому мы можем зашифровать этот симметричный ключ с помощью открытого ключа получателя и сохранить результат в начале неизменяемой записи. Затем можем зашифровать содержимое сообщения с помощью симметричного ключа и сохранить этот зашифрованный блок в конце записи. Получатель может проделать обратный процесс, используя свой закрытый ключ, чтобы раскрыть симметричный ключ и извлечь его из случайной подложки. Затем он может расшифровать окончание записи, чтобы раскрыть тело сообщения, как показано на рис. 7.9.

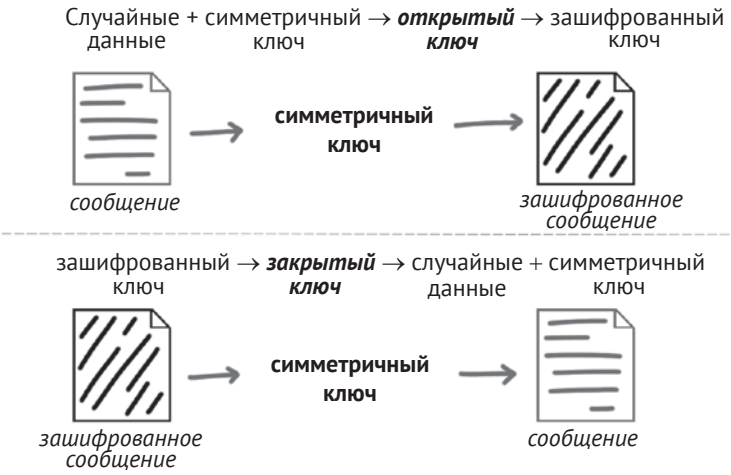


**Рис. 7.9.** Отправитель шифрует симметричный ключ с помощью открытого ключа получателя

Промежуточные участники этого обмена не смогут легко расшифровать запись без закрытого ключа получателя. Нам удалось обеспечить конфиденциальность, когда между участниками находятся недоверенные узлы. В отличие от шифрования во время транспортировки – что раскрывает содержимое сообщения посредникам, – мы зашифровали его *до* передачи. Даже если сторонний узел не предпримет никаких дополнительных мер предосторожности, мы защитили сообщение в состоянии покоя.

## Шифрование исторических фактов

Протокол, описанный ранее, является популярным механизмом для обмена частными сообщениями через ненадежные носители. Он является основной процедурой в протоколе OpenPGP[1], используемом для обмена частной электронной почтой. Он обычно применяется к сообщениям, хранящимся в распределенных файловых системах, таких как IPFS, InterPlanetary File System. В этих и других сценариях протокол имеет ограничение – он защищает только содержимое сообщения, а не метаданные, окружающие сообщение. Хотя злоумышленник не сможет прочитать содержание электронной почты OpenPGP, он сможет определить отправителя и получателя. Это просто следствие того, что сторонним узлам необходимо знать, как маршрутизировать сообщения. Когда этот протокол используется в рамках исторической модели, действует то же ограничение.

В исторической модели шифруется только тело конфиденциального факта. Тип и, что более важно, предшественники этого факта не шифруются. Причина в том, что сторонние узлы должны иметь возможность выполнять запросы и возвращать конфиденциальные факты. Если предшественники были зашифрованы, то сторонний узел не смог бы определить, какие факты были преемниками, искомыми в запросе.

[1] *J. Callas, et al.* IETF Network Working Group Request for Comments 4880. November 2007.

По аналогичной причине идентификация конфиденциального факта основывается на хеше его зашифрованного представления, а не оригинального тела. Если впоследствии записываются другие факты, использующие конфиденциальный факт в качестве предшественника, промежуточные стороны должны понимать эту взаимосвязь. Они должны иметь возможность получить хеш записи без ее расшифровки.

## Ограничьте распространение конфиденциальных фактов

Поскольку типы и отношения предшественников свободно видны промежуточным сторонам, мы все равно должны быть осторожны с теми, с кем мы делимся зашифрованными конфиденциальными сообщениями. Мы можем избегать общих распределенных реестров таких как блокчейн, которые полагаются на публичную проверку метаданных, для того чтобы функционировать. Мы можем ограничить круг коллег, с которыми напрямую обмениваемся зашифрованными фактами, только теми, которым можем доверять настолько, чтобы не выводить значение из отношения предшественник/преемник. Мы даже можем распределить наше доверие между несколькими посредниками, чтобы ни один из них не имел полной картины наших взаимодействий с другими сторонами.

Чтобы предотвратить распространение конфиденциальных фактов дальше, чем это необходимо, эти посредники должны иметь набор правил, в соответствии с которыми им разрешается раскрывать эту информацию. В отличие от правил авторизации, рассмотренных ранее, мы не можем сами выполнять эти правила. Мы должны верить, что посредники, с которыми мы решили работать, соблюдают их при каждом запросе. Правила указывают им, кому разрешено получать определенные факты.

### Правила распространения

Если правило авторизации говорит нам о том, кому разрешено использовать факт, то правило распространения говорит о том, кому разрешено выполнять запрос. Мы начинаем с определения разрешенных запросов. Они могут быть выражены с помощью языка запросов Factual, как мы уже показали на протяжении всей книги. Дайте каждому узлу сервера список разрешенных Factual-запросов. Любой клиент, выдающий запрос, не включенный в этот список, немедленно отклоняется.

Следующий шаг – определить разрешенные начальные точки для этих запросов. Каждый Factual-запрос имеет одну начальную точку. Обычно он сопоставляет преемников этого начального факта, а затем, возможно, зигзагообразно перемещается по графу. Правило распространения выражает набор принципалов, которым разрешено начинать запрос с этой точки. Как и в случае с правилом авторизации, это делается с помощью запроса.

Предположим, что мы написали следующий запрос для личных сообщений, отправленных человеку:

```
query privateMessagesToRecipient(r: User) {
  match m: PrivateMessage where m.recipient = r
}
```

Достаточно строгий узел сервера не будет принимать никаких запросов до тех пор, пока ему не будет предоставлена такая возможность. Он не будет принимать никаких запросов на факты `PrivateMessage`, пока ему не будет предоставлен предыдущий запрос и следующее правило распространения:

```
distribute privateMessagesToRecipient for r: User {
  match r
}
```

Это правило максимально простое, но оно демонстрирует важные моменты. Оно идентифицирует запрос, в данном случае тот, который был определен ранее. Оно начинается с того же факта, что и запрос, который оно контролирует. Исходя из этой отправной точки, оно подбирает принципалов, которым разрешено выполнять запрос. Это конкретное правило распространения позволяет только получателю запрашивать личные сообщения.

Каждый запрос имеет свою каноническую форму. Сервер может использовать ее, а также тип начального факта для поиска соответствующих правил распространения. Если узел сервера получает запрос, который не совпадает с тем, который ему было разрешено распространять, он отклонит запрос. Если клиент не может доказать, что он действует от имени хотя бы одного принципала, соответствующего правилу распределения, то сервер отклонит запрос. Если эти две проверки пройдут, то сервер выполнит запрос и вернет результаты. Сервер может быть не в состоянии интерпретировать содержание полученных фактов, но он сделает все возможное, чтобы ограничить их распространение только теми, кому это должно быть доступно.

## Доказательства

Доказательство того, что клиент действует от имени принципала, не является тривиальным вопросом. То, как это выполняется, зависит от того, откуда поступает запрос – от клиента или от другого сервера. Если пользователь вошел в систему как клиент, у него будет маркер авторизации, которым он может поделиться с сервером. Маркер должен быть выдан службой маркеров безопасности (security token service – STS), которой доверяют и клиент, и сервер. Клиент предоставляет свои учетные данные службе STS, и она подписывает маркер. В этом сценарии сервер, который клиент использует напрямую, часто выступает в качестве хранилища ключей. Он будет хранить закрытый ключ пользователя, чтобы пользователь мог войти в систему с любого устройства. Этот сервер должен быть полностью под контролем пользователя.

Если связь между клиентом и сервером может быть авторизована с помощью маркера, то связь между серверами не может. Сервер инициирует соединение с другим сервером без вмешательства пользователя. Пользователь не должен предоставлять учетные данные для STS и генерировать маркер. Более того, у двух серверов может не быть общей STS, которой они оба доверяют. Маркер

безопасности не обеспечит удовлетворительного доказательства того, что запрос выполняется от имени данного пользователя.

Чтобы обеспечить удовлетворительное доказательство, запрашивающий сервер должен использовать закрытый ключ принципала, инициирующего запрос. Учитывая, что этот сервер также может быть хранилищем ключей, этот процесс может быть выполнен без вмешательства пользователя. Протокол начинается с того, что запрашивающий сервер вызывает запрос из определенной начальной точки и идентифицирует открытый ключ принципала. Если целевой сервер определяет, что идентифицированному принципалу разрешено выполнить запрос, он отвечает случайным образом сгенерированным вызовом. Запрашивающий сервер отвечает на вызов таким образом, чтобы доказать, что он владеет закрытым ключом. Хитрость заключается в том, что он не должен просто использовать закрытый ключ и вернуть ответ. Сделать это – значит позволить серверу использовать дайджест сообщения в качестве вызова и таким образом сгенерировать действительную подпись или выполнить атаку «человек посередине»[1] и выдать действительный ответ на вызов. Вместо этого необходимо использовать протокол *доказательства нулевого знания*. Такой протокол доказывает, что запрашивающий сервер имеет закрытый ключ, но не позволяет целевому серверу получить какие-либо знания о нем. Дэвид Чаум и его коллеги[2] приводят несколько протоколов для доказательства того, что запрашивающий сервер знает дискретный логарифм (обратный RSA-ключу) заданного значения без раскрытия этого значения.

## Атаки и контрмеры

Однако ни одна из этих мер предосторожности не защищает нас от третьих лиц, которые злонамеренно, по неосторожности или по принуждению передают нашу информацию другим лицам. Для этого нам могут понадобиться дополнительные меры предосторожности, основанные на чувствительности метаданных. Дополнительные меры включают:

- периодическую смену пар ключей для маскировки личности участника;
- смешивание одноразовых маркеров со случайными пользователями при обмене ключами;
- генерацию новой пары ключей и, следовательно, нового факта принципала для каждого взаимодействия;
- хранение таблиц сопоставления идентификационных данных в автономном режиме или на частном управляемом сервере;
- генерацию множества бессмысленных фактов между не связанными друг с другом предшественниками, чтобы скрыть сигнал в шуме.

---

[1] Атака «человек посередине» (англ. Man in the middle (MITM)) — вид атаки в криптографии и компьютерной безопасности, когда злоумышленник тайно ретранслирует и при необходимости изменяет связь между двумя сторонами, которые считают, что они непосредственно общаются друг с другом. – *Прим. перев.*

[2] *Chaum David, Evertse Jan-Hendrik, van de Graaf Jeroen (1987)*. An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations. Advances in Cryptology – EuroCrypt '87: Proceedings. Lecture Notes in Computer Science.

Такие контрмеры необходимы не для каждой области. Но когда связи между фактами имеют ценность, а промежуточным сторонам нельзя полностью доверять, тогда дополнительная осторожность оправдана.

Еще один вектор атаки необходимо учитывать при хранении зашифрованных фактов на недоверенных промежуточных узлах. Данные, находящиеся в состоянии покоя, подвержены автономным атакам. Онлайн-сервисы блокируют неудачные попытки доступа к данным, чтобы предотвратить атаки «грубой силы». Но когда в распоряжении злоумышленника есть зашифрованные данные, он может выполнить столько попыток, сколько позволит его вычислительная мощность. Такая атака против современных криптографических алгоритмов будет дорогостоящей. Но решительный злоумышленник с достаточно ценной конечной целью может захотеть и – в конечном итоге – обнаружить симметричный ключ.

Есть несколько вещей, которые могли бы облегчить его работу. Например, чем чаще симметричный ключ используется повторно, тем больше образцов для работы у злоумышленника. И чем длиннее зашифрованное сообщение, тем больше вероятность того, что он обнаружит пригодный для эксплуатации образец. Поэтому, чтобы защитить информацию от атак в автономном режиме, делайте все наоборот:

- используйте сильный генератор случайных чисел для создания симметричных ключей;
- используйте каждый симметричный ключ только один раз;
- делайте сообщения как можно короче;
- заполняйте сообщения большим количеством случайных данных;
- генерируйте дополнительные бессмысленные факты, чтобы скрыть ценные.

Даже при соблюдении всех этих мер предосторожности вы должны исходить из того, что криптосистемы, которые мы используем сегодня, рано или поздно устареют. Это было обнаружено в отношении алгоритмов прошлого, и у нас нет оснований полагать, что будущее не выявит слабые места в сегодняшней технологии. На самом деле есть основания полагать, что квантовые вычисления когда-нибудь сделают все сегодняшние асимметричные алгоритмы неэффективными. Возможно, все, что нужно сделать решительному злоумышленнику, – это подождать.

Мой единственный совет по смягчению этой проблемы – убедиться, что ценность вашей информации снижается с течением времени. Если вы обмениваетесь платежной информацией, убедитесь, что срок действия этих инструментов истечет до того, как шифрование будет взломано. Если вы сотрудничаете над секретным планом, убедитесь, что вы выполните его до того, как станет слишком поздно. Если у вас есть что-то, что вы хотите сохранить в тайне навсегда, не шифруйте это и не передавайте третьей стороне.

## Секретность

Используя PKI, правила распространения и контрмеры, мы добились определенного уровня конфиденциальности. Теперь мы можем отправить сообщение определенному получателю через недоверенные узлы и иметь некоторую гарантию того, что содержимое останется конфиденциальным

в течение некоторого времени. Эта форма конфиденциальности хорошо работает при общении с одним человеком. Однако многие наши совместные проекты связаны с группами. Теперь мы обобщим эти методы, чтобы группа могла общаться тайно.

Начнем с определения общего *рабочего пространства*, в котором взаимодействуют сотрудники. Это может быть проект, содержащий рабочие элементы, планы и ресурсы. Или это может быть отдел, например бухгалтерия, где хранятся платежные ведомости, кредиторская задолженность и активы. Наша цель – контролировать доступ в этом рабочем пространстве. Мы уже продемонстрировали использование правил авторизации для предоставления полномочий на создание новых фактов в рабочем пространстве. Теперь мы расширим нашу стратегию конфиденциальности, чтобы сохранить эти факты в тайне, видимыми только для тех, кто приглашен в рабочее пространство.

## Общий симметричный ключ

Чтобы сохранить данные в рабочем пространстве в тайне, мы создадим общий симметричный ключ. Любой человек, владеющий этим ключом, сможет расшифровать содержимое фактов в пределах пространства. Поэтому у нас должен быть протокол для обмена симметричным ключом с другими членами рабочего пространства в рамках исторической модели. Мы должны сделать это, не раскрывая его ненадежным третьим сторонам, которые хранят и пересылают факты.

Рабочее пространство в исторической модели имеет форму факта. У него есть только один предшественник, создатель, и только одно поле, его идентификатор. Когда пользователь первоначально создает рабочее пространство, он генерирует случайный симметричный ключ. Этот симметричный ключ не хранится в рабочем пространстве. Вместо этого он передается каждому участнику по отдельности.

Чтобы поделиться симметричным ключом с другими участниками, создатель посылает приглашения. Приглашение является преемником факта рабочего пространства с указанием получателя в качестве предшественника. Его единственным полем является общий симметричный ключ. Создатель рабочего пространства посылает первое приглашение самому себе. Симметричный ключ упаковывается в случайную подложку и шифруется с помощью своего собственного открытого ключа. Теперь общий ключ хранится в модели таким образом, что только создатель может его расшифровать. Он может отправлять приглашения другим участникам сразу или по мере их присоединения к команде.

### Секретный канал для обсуждения

Чтобы продемонстрировать этот протокол, давайте рассмотрим секретный канал, на котором сотрудники могут публиковать сообщения. Сам канал – это не что иное, как факт, имеющий идентификатор. Приглашения – это факты-преемники, посылаемые участникам. Модель показана на рис. 7.10.

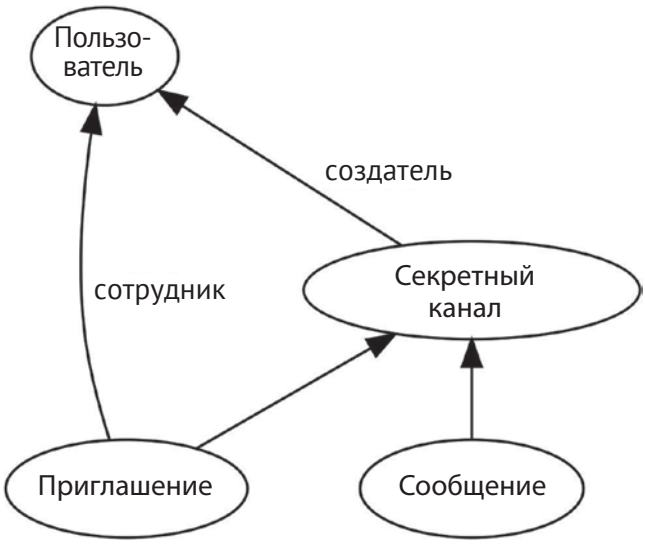


**Рис. 7.10.** Секретный канал, в который приглашаются сотрудники

Модель выражается на языке Factual следующим образом:

```
fact User {
  publicKey: string
}
fact SecretChannel {
  creator: User
  identifier: guid
}
fact Invitation {
  secretChannel: SecretChannel
  collaborator: User
  sharedKey: string
}
fact Message {
  secretChannel: SecretChannel
  body: string
}
```

## Создание секретного канала

Алиса хочет создать секретный канал, чтобы общаться с Бобом и Чарли. Она начинает с генерации глобального уникального идентификатора для канала, а также случайного симметричного ключа. С их помощью она создает факт «Секретный канал» (SecretChannel) и три факта «Приглашение» (Invitation). Одно из приглашений адресовано ей самой. Полученные факты показаны на рис. 7.11.

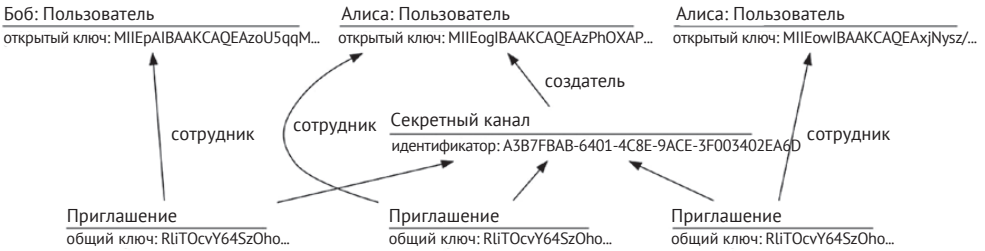


**Рис. 7.11.** Алиса создала частный канал и пригласила Боба и Чарли

Несмотря на то что на предыдущей диаграмме показано содержание фактов приглашения в открытом тексте, каждый из них зашифрован с использованием открытого ключа соответствующего сотрудника. Каждый сотрудник может индивидуально расшифровать свое собственное приглашение. При этом раскрывается общий симметричный ключ.

## Командные правила распространения

Даже если содержимое сообщений зашифровано с помощью общего симметричного ключа, все же разумно ограничить распространение этих фактов. Если предположить, что промежуточные сторонние узлы, которые мы решили использовать, будут хорошо себя вести, они будут следовать правилам, которые мы определяем для распространения. Вот запрос на секретные сообщения и соответствующее правило распространения:

```
query messagesInSecretChannel(sc: SecretChannel) {
  match m: Message where m.secretChannel = sc
}
distribute messagesInSecretChannel for sc: SecretChannel {
  match i: Invitation where i.secretChannel = sc
  then u: User where i.collaborator = u
}
```

Это правило распространения позволяет серверу отвечать на запросы сотрудников, у которых есть приглашение в этот секретный канал. Запросы от других пользователей будут отклонены, чтобы они не смогли выполнить автономный криптоанализ сообщений для обнаружения общего симметричного ключа.

Сравните правило распространения с соответствующим правилом авторизации:

```
authorize m: Message {
  match i: Invitation where i.secretChannel = m.secretChannel
  then u: User where i.collaborator = u
}
```

Это правило разрешает каждому сотруднику размещать сообщения в секретном канале. Оно использует то же приглашение, что и правило распространения. Однако это не обязательно. Мы можем определить другой тип приглашения, называемый `ReadOnlyInvitation`. Определив правило распространения, основанное на этих типах приглашений, но не определяя аналогичное правило авторизации, мы можем предоставить некоторым пользователям доступ к сообщениям только для чтения.

## Ограничение области применения общего ключа

Точно так же, как мы хотим ограничить распространение зашифрованных фактов, мы хотим ограничить число фактов, которые шифруются с помощью одного и того же симметричного ключа. Повторное использование симметричного ключа дает потенциальному злоумышленнику больше примеров шифрованного текста для анализа. При отправке факта одному получателю мы смогли сгенерировать отдельный симметричный ключ для каждого сообщения (обычно называемый *сеансовым ключом*). К сожалению, этот же механизм не будет работать для рабочих пространств с большим количеством участников. Сеансовый ключ должен быть зашифрован с помощью открытых ключей каждого сотрудника по очереди. Некоторые из этих участников могут даже не присоединиться к команде на момент создания зашифрованного факта.

Поэтому мы должны прибегнуть к повторному использованию общего симметричного ключа для всех секретных фактов в пределах рабочего пространства. При этом мы еще раз сталкиваемся с проблемой отмены ключа. Мы хотим быть уверены, что защитим текущую работу оставшихся, после того как бывший сотрудник покидает команду. Мы можем достаточно легко определить факт отмены приглашения (InvitationRevoke) как преемника приглашения. При наличии такого факта, который не существует в правиле распространения, промежуточные узлы больше не будут распространять защищенные факты бывшему сотруднику. Но этот человек все еще имеет доступ к общему симметричному ключу. Если он сможет заставить промежуточный узел отдать свой кеш зашифрованных фактов, у него будут инструменты для их расшифровки.

### Когорты

Одним из способов решения проблемы отмены является переход всех оставшихся сотрудников к новому общему ключу и выдача им новых приглашений. Бывший сотрудник не будет иметь доступа к этому ключу. Таким образом, хотя он сможет продолжать расшифровывать информацию, которую он видел, будучи членом проекта, у него не будет доступа к новой информации.

Переход сотрудников на новый общий ключ – это неудобство, но с ним можно справиться. Все факты, созданные ими до перехода, будут зашифрованы с помощью старого общего ключа. Сотрудники захотят создать новые факты, используя новый ключ, но при этом иметь возможность ссылаться на старые факты как на предшественников. Работа должна продолжаться непрерывно.

Чтобы смоделировать это решение, мы вводим уровень непрямолинейности. Между рабочим пространством и последующей работой мы вводим *когорту*. Этот факт представляет группу сотрудников, которые в одно и то же время поделились симметричным ключом. Все последующие факты в рабочем пространстве связаны с когортой, которая сообщает нам, каким симметричным ключом они были зашифрованы. Модифицируя модель секретного канала на рис. 7.10, когорта вписывается в модель, показанную на рис. 7.12.


**Рис. 7.12.** Сотрудников приглашают присоединиться к когорте

## Периоды

Другим решением проблемы отзыва является установление часов в области. Используйте часы в качестве когорты, как было показано ранее. Каждый период времени получает свой собственный общий симметричный ключ. Все оставшиеся участники получают новые приглашения в начале каждого нового периода. Бывшие сотрудники отпадают. Хотя это оставляет окно для атаки открытым до начала следующего периода, его преимущество в том, что оно тесно связано с областью. Ресторан может определить период как одну дату ведения бизнеса. Школа может определить его как один семестр. И в таких областях границы когорты часто совпадают с часами. *Шаблон «Период»* более подробно рассматривается в главе 8.

Мы собрали набор инструментов, которые могут помочь нам определить доступ к исторической модели, не полагаясь на административный орган. Начиная с основ PKI, мы использовали пары открытый/закрытый ключ для идентификации принципалов, таких как пользователи или автономные системы. Используя закрытый ключ для подписи дайджеста, мы можем доказать автор-

ство каждого факта. Затем каждый узел может запустить правила авторизации, чтобы проверить полномочия автора на выполнение требуемого действия. А используя открытый ключ для шифрования симметричного ключа, мы можем сохранить конфиденциальность фактов, предназначенных как для одного человека, так и для команды. Вся информация, необходимая для выполнения этих правил, находится внутри самой модели. Это дает каждому узлу автономию для безопасной работы в коллективной среде без необходимости полагаться на наличие доверенной третьей стороны.

# Глава 8

# Шаблоны

Теперь у вас есть набор инструментов, с помощью которых вы можете создавать системы, которые от природы устойчивы и надежны. Возможно, вы даже имеете представление о том, как применить эти инструменты для решения бизнес-проблем, с которыми сталкиваются ваши клиенты. Но могут остаться некоторые пробелы.

В этой главе мы систематически рассмотрим, как применять правила исторического моделирования для решения реальных проблем. Начиная с общих проблем, мы выведем исторические решения. Затем, используя аналитические инструменты, которые мы разработали в предыдущих главах, изучим последствия этих решений. В результате получим каталог шаблонов, на которые сможем ссылаться при построении новых моделей. Этот каталог не будет всеобъемлющим, но он обеспечит хорошую основу для поиска новых решений проблем, с которыми вы столкнетесь в будущем.

## Структурные шаблоны

Значительная часть программного обеспечения, которое мы пишем для бизнес-клиентов, относится к категории *форм для работы с данными*, иногда известным как *CRUD*. Это вид программного обеспечения, которое предоставляет пользователю возможность создавать, читать, обновлять и удалять (create, read, update, and delete) объекты. Это не очень привлекательная работа, но она должна выполняться.

Реляционные модели и модели гипермедиа, похоже, были разработаны с учетом CRUD-приложений. Базы данных связывают эти четыре операции с четырьмя основными командами – INSERT, SELECT, UPDATE и DELETE. Гипермедийные приложения, использующие POST, GET, PUT и DELETE, кажется, отражают основные операции CRUD.

Но реализация операций CRUD в исторической модели не столь ясна. Наиболее очевидный диссонанс заключается в том, что историческая модель не допускает обновления или удаления. Пользователь хочет выполнить эти операции, но модель не позволяет их выполнять. Поэтому мы должны найти способ симулировать эти операции.

Там, где реляционное и гипермедийное моделирование обеспечивают прямой аналог CRUD-операций, исторические модели требуют немного больше внимания. Чтобы примирить различия между потребностями CRUD и возможностями исторического моделирования, давайте пройдемся по понятиям CRUD по очереди. Мы построим шаблоны, которые позволят моделировать каждое из них в рамках строгих правил неизменяемости.

## Сущность

**Значение** Представляет создание сущности.

В приложении с формами для работы с данными пользователю нужна возможность создавать новые объекты. Обычно он нажимает кнопку и получает форму. После того как он заполнит ее и нажмет другую кнопку, система создает сущность и присваивает ей идентичность.

Идентичность строки в реляционной базе данных иногда создается с помощью автоинкрементного идентификатора. Эта стратегия не подходит для исторической модели, так как в этом случае она будет опираться на идентификатор, зависящий от местоположения. Различные узлы могут генерировать один и тот же идентификатор для разных сущностей. Необходим идентификатор, не зависящий от местоположения.

Еще одним моментом различия является инициализация новой сущности. Реляционные базы данных имеют операторы INSERT, которые устанавливают во все столбцы новой строки значения по умолчанию или заданные значения. Но в исторической модели не имеет смысла при создании сущности инициализировать ее свойства. Какая-то будущая операция захочет изменить эти свойства. Сам исторический факт неизменяем, поэтому использовать его для хранения начальной версии набора изменяемых свойств неудобно. Это сделает начальную версию чем-то отличным от будущих обновлений. Это также сделает эти начальные значения частью идентичности сущности даже после того, как они были впоследствии изменены.

Шаблон *Entity* (Сущность) фокусируется на построении независимой от местоположения идентичности и избегает инициализации изменяемых свойств.

### Структура

Сущность – это исторический факт, который содержит только идентифицирующую информацию. Она содержит естественный ключ, GUID, временную метку или некоторую комбинацию этих и других идентификаторов, не зависящих от местоположения.

```
fact Entity {
  identifier1: type
  identifier2: type
}
```

Выдача такого факта эквивалентна созданию сущности. Он представляет собой как идентичность, так и создание самой сущности.

## Пример

Товар может быть представлен фактом, который просто фиксирует SKU (единицу хранения на складе):

```
fact Product {
  sku: string
}
```

Описание, цена, количество в наличии, статус заказа и другие свойства товара не хранятся в факте. Эти свойства можно изменять. Факт является неизменяемым. Он представляет собой идентификатор товара и факт его создания.

## Последствия

Сущность должна использовать независимую от местоположения идентификацию. Она не может использовать автоинкрементные идентификаторы, URL или любой другой идентификатор, зависящий от местоположения.

Сущность не содержит изменяемых свойств. Любые изменяемые свойства, которые должны быть связаны с сущностью, применяются с последующим фактом.

Если два узла создают сущности с одинаковыми идентификаторами, то это одна и та же сущность. Узлы могут не знать друг о друге в момент создания, но любые узлы, которые узнают об этих двух сущностях, будут считать, что они одинаковые.

Если информация аудита, например создатель, местоположение или время создания, добавляется к сущности, то она становится частью ее идентичности. Выбор сделать эту информацию частью идентификатора – это один из способов обойти предыдущее следствие – то, что две сущности с одинаковыми идентификаторами являются одной и той же сущностью. Делайте это только в том случае, если это важно для модели. В противном случае держите информацию для аудита за пределами самих фактов.

## Связанные шаблоны

Сущность, включающая идентификатор своего родителя, соответствует шаблону *Ownership* (Владение).

Хотя факт сущности не может быть удален, шаблон *Delete* имитирует удаление сущности.

Изменяемые свойства не включаются в факт сущности. Вместо этого они присоединяются с помощью шаблона *Mutable Property*.

## Владение

**Значение** Представляет строгую иерархию между сущностями.

Нередко модель имеет строгую иерархию. В доменно-ориентированном дизайне[1] такая структура называется *агрегатом*. В реляционной модели это особый вид отношения «один ко многим», где каждый ребенок имеет только одного родителя, иногда с ограничением каскадного удаления. Такой вид строгого владения часто называют отношением «*родитель–ребенок*».

Идентификаторы нередко отражают строгую принадлежность сущности. В REST идентификаторы ресурсов имеют структуру пути, которая показывает, какие из них содержатся внутри других. В файловых системах каждая папка существует строго внутри одной родительской папки. Путь к папке включает идентификатор родительской папки.

В исторической модели нет строгой необходимости идентифицировать одного предшественника как владельца преемника. Тем не менее такие отношения часто встречаются в проблемной области. Поэтому мы будем обычно представлять эти особые отношения с помощью условных обозначений.

Шаблон *Ownership* документирует строгое отношение «родитель–потомок» между преемником и одним из его предшественников.

### Структура

Родитель сущности представляется как предшественник ее идентифицирующего факта, как показано на рис. 8.1.


**Рис. 8.1.** Каждый потомок принадлежит только одному родителю

Предшественник родителя перечисляется первым в полях дочернего факта. Он предшествует даже идентификаторам дочернего факта. Хотя это соглашение не заставляет родителя вести себя иначе, чем любой другой предшественник, это недорогой способ документирования желаемого отношения владельца.

[1] *Eric Evans.* Domain-Driven Design: Tacking Complexity In the Heart of Software. AddisonWesley Longman Publishing Co., Inc. Boston, MA, USA ©2003. ISBN:0321125215.

```
fact Child {
  parent: Parent
  identifier: type
}
```

Иногда дочернему факту присваивается имя, включающее имя родительского факта. Это не является строгим соглашением и может быть нарушено, если связь очевидна или имена становятся слишком длинными.

```
fact Owner {
  identifier: type
}
fact OwnerItem {
  owner: Owner
  itemIdentifier: type
}
```

Запросы к дочерним сущностям часто начинаются с их родителя. Для данного родителя запрос возвращает всех потомков.

```
query childrenOfParent(p: Parent) {
  match c: Child where c.parent = p
}
```

Однако иногда возникают ситуации, когда запрос ссылается на дочерний объект с каким-либо другим отношением, отличным от родительского. Когда это происходит, запрос должен включать условие, что родительская сущность не была удалена.

```
query childrenRelatedTo(r: Relation) {
  match c: Child where c = r.relatedChild
       such that not exists d: ParentDeleted where d.parent = c.parent
}
```

Правила авторизации часто выражаются по границам владения. Чтобы дать кому-то возможность создавать дочерние сущности, мы назначаем повышенные разрешения для родителя. Эти правила авторизации должны быть определены явно, поскольку право собственности само по себе не является неявной частью исторической модели.

```
authorize c: Child {
  match a: Authorization where a.object = c.parent.creator
       such that not exists r: Revocation where r.authorization = a
  then u: User where u = a.subject
}
```

## Пример

Заказ принадлежит строго той компании, которой он сделан. Каждый заказ также имеет отличительный атрибут `GUID`, чтобы отделить его от других заказов для той же компании.

```
fact Order {
  company: Company
  orderGuid: guid
}
```

Этот факт не использует условное имя `CompanyOrder`. Префикс владельца в данном случае просто удлиняет имя, не имея никакой реальной ценности. Можно предположить, что многие из сущностей в этой системе принадлежат компании.

Заказ будет содержать линейные позиции. По соглашению, мы даем дочернему факту составное имя, которое включает имя родителя.

```
fact OrderLine {
  order: Order
  createdAt: datetime
}
```

Этот факт *соответствует* соглашению об именовании, поскольку в противном случае нельзя было бы предположить, что строка принадлежит заказу. Возможно, система также моделирует счета-фактуры с ассоциированными с ними строками счета-фактуры (InvoiceLines).

И `Order`, и `OrderLine` следуют соглашению о перечислении владельца первым среди полей, даже перед идентификаторами дочерних сущностей.

Время `createdAt` позволяет различать строки заказов в рамках одного заказа. Временные метки сами по себе не являются достаточными идентификаторами, но в сочетании с другими идентификаторами, такими как родительская сущность в данном случае, они могут быть эффективными. Предполагается, что строки заказов будут добавляться одним пользователем с одного узла и что количество строк заказов будет очень небольшим.

Также обратите внимание, что `createdAt` – это временная метка, полученная в момент фактического создания строки заказа. Это время на рабочей станции пользователя. Это не время, в которое веб-сервер или другой нижестоящий узел узнал о строке заказа. Это время, когда пользователь физически нажал на кнопку в браузере или клиентском приложении.

Полученная модель показана на рис. 8.2.

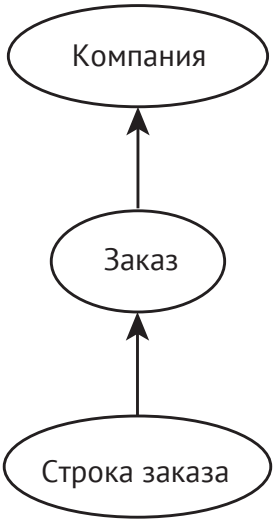


**Рис. 8.2.** Строка заказа принадлежит одному заказу, который, в свою очередь, принадлежит одной компании

### Последствия

Идентичность родителя является частью идентичности потомка. Поскольку предпосылки неизменяемы, потомки не могут быть перемещены к другому родителю. Владение не подлежит передаче.

Родитель должен существовать до того, как может быть создан потомок. Владение не применяется к коллекции отдельных сущностей, которые впоследствии группируются после их создания.

Шаблон *Ownership* (Владение) поощряет многопользовательскую работу. Идентичность корневого владельца, как правило, становится частью идентичности большинства других сущностей. В противном случае существует возможность заражения от соседних узлов, находящихся под контролем других организаций, особенно если они склонны генерировать перекрывающиеся идентификаторы.

### Связанные шаблоны

Если необходимо передать владение, рассмотрите вместо этого шаблон Membership (Членство).

Шаблон Ownership (Владение) является частным случаем шаблона Entity (Сущность), в котором идентификаторы включают идентичность владельца.

## Удаление

**Значение** Представляет удаление сущности.

Исторические факты неизменны, они не могут быть ни изменены, ни уничтожены. Однако удаление является обычной частью бизнес-приложений. Поэтому удаление моделируется путем добавления нового факта.

Обычной практикой в реляционной базе данных является включение удаленного столбца в таблицу. Это булев флаг, который устанавливается, когда строка должна быть удалена. Все запросы включают предложение WHERE, которое исключает удаленные строки. Такая процедура известна как *мягкое удаление*.

Шаблон *Delete* удаления исторических фактов, однако, немного отличается. Установка флага является модификацией. Историческая модель не допускает модификаций. Поэтому удаление не может быть смоделировано путем установки флага. Оно должно быть представлено как создание нового факта.

## Структура

Факт удаления принимает сущность, которую он удаляет, в качестве предшественника. По соглашению, именем факта удаления является `Deletion`, добавляемое к имени сущности.

```
fact Entity {
  identifier: type
}
fact EntityDeletion {
  entity: Entity
}
```

Любой запрос к этому предшественнику должен включать предложение `not exists`, которое исключает сущности, которые были удалены. Например, если у предшествующей сущности был владелец, то запрос для дочерних сущностей будет выражен следующим образом:

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
        such that not exists ed: EntityDeletion where ed.entity = e
}
```

## Пример

В предыдущем примере к заказу были добавлены строки заказа. Если бы приложение позволяло пользователю удалять строки из заказа, оно бы представило их как `OrderLineDeletions`, как показано на рис. 8.3.


**Рис. 8.3.** Строка заказа была удалена из заказа

Запрос строк в заказе должен исключить все удаленные строки:

```
query linesInOrder(o: Order) {
  match ol: OrderLine where ol.order = o
       such that not exists old: OrderLineDeletion where old.orderLine = ol
}
```

## Последствия

Если факт удаления не имеет идентификаторов, позволяющих отличить его от других удалений той же сущности, то сущность может быть удалена только один раз. Чтобы разрешить восстановление сущности в будущем, добавьте отличительный идентификатор. В большинстве случаев достаточно метки времени.

## Связанные шаблоны

Если удаление должно быть обратимым, рассмотрите возможность использования шаблона *Restore*.

## Восстановление

**Значение** Отмена предыдущего удаления.

Почти каждое приложение, позволяющее удалять данные, использует один из двух методов для смягчения последствий случайного удаления. Более распространенным является подтверждение. В других приложениях предлагается способ восстановления удаленной сущности.

Восстановление может начинаться с корзины, в которой перечислены все удаленные сущности. Или оно может быть доступно только сразу после удаления в виде отмены.

### Структура

Факт восстановления ссылается на предыдущее удаление. По соглашению, он добавляет слово `Restoration` к имени сущности. Удаление имеет дополнительный идентификатор, обычно это метка времени. Восстановление не имеет дополнительных идентификаторов.

```
fact Entity {
  identifier: type
}
fact EntityDeletion {
  entity: Entity
  deletedAt: timestamp
}
fact EntityRestoration {
  deletion: EntityDeletion
}
```

Любой запрос к сущности включает условие `not exists` для удалений, которое, в свою очередь, имеет условие `not exists` для восстановлений. Если у предыдущей сущности был владелец, то запрос для дочерних сущностей выглядел бы следующим образом:

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
       such that not exists ed: EntityDeletion where ed.entity = e
              such that not exists er: EntityRestore where er.deletion = ed
}
```

Если пользователю предлагается корзина, из которой можно восстановить сущности, она отображает результаты запроса, в котором существует удаление. Обратите внимание, что это точно такой же запрос, как и предыдущий, за исключением того, что `not` было исключено из `exists ed: EntityDeletion`.

```
query entitiesInRecycleBin(o: Owner) {
  match e: Entity where e.owner = o
       such that exists ed: EntityDeletion where ed.entity = e
              such that not exists er: EntityRestore where er.deletion = ed
}
```

Симметрия этих запросов делает действия по удалению и восстановлению атомарными. Создание удаления одновременно добавляет сущность в корзину и удаляет ее из приложения. Последующее создание восстановления одновременно удаляет сущность из корзины и вновь интегрирует ее в приложение.

## Пример

В предыдущем примере мы рассмотрели модель, которая поддерживала удаление строк из заказа. Чтобы поддержать восстановление удаленных строк обратно в заказ, мы добавим факт `OrderLineRestoration` в модель, как показано на рис. 8.4.

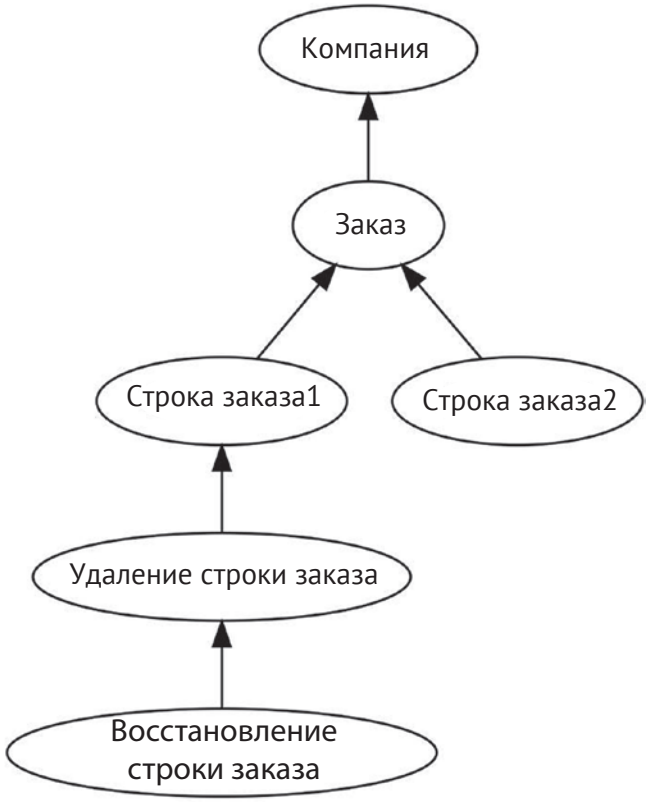


**Рис. 8.4.** Строка заказа, ранее удаленная из заказа, была восстановлена

В примере с удалением для `OrderLineDeletion` не требовалось никакого дополнительного идентификатора. Однако, чтобы поддержать восстановление, `OrderLineDeletion` теперь имеет поле временной метки.

### Последствия

Если факт удаления не имеет дополнительного идентификатора, например временной метки, то сущность может быть удалена и восстановлена только один раз. После этого невозможно будет удалить сущность снова. Второе удаление не будет отличаться от первого, которое было восстановлено. Такое поведение почти наверняка нежелательно. Поэтому временная метка фактически является требованием.

### Связанные шаблоны

*Restore* является расширением шаблона *Delete*. Если сущность может быть восстановлена под новым именем без потери корректности, тогда следует использовать более простой шаблон *Delete*. Это часто предпочтительнее, когда сущность не имеет изменяемых свойств и не участвует в рабочем процессе. Но если существуют свойства, рабочий процесс или возможны какие-либо другие преемники, то более подходящим будет шаблон *Restore*.

## Членство

---

**Значение** Добавляет сущности в группы, которые могут быть изменены с течением времени.

---

В то время как строгое владение не позволяет сущностям переходить от одного родителя к другому, некоторые бизнес-приложения требуют такой гибкости. Сотрудник может работать в одном отделе, а затем быть переведен в другой. Проект может быть частью одного портфолио при его инициировании, но затем быть переведен в другое. Эти отношения группировки – не строгое владение, а более гибкое членство.

### Структура

Отношение между членом и группой представлено как факт, имеющий как члена, так и группу в качестве предшественников. Факт членства имеет дополнительный идентификатор, обычно метку времени, который позволяет удалять и добавлять члена в группу с течением времени.

По соглашению член перечисляется первым среди полей членства, перед группой. Оба поля появляются перед дифференцирующим идентификатором.

```
fact Group {
  identifier: type
}

fact Member {
  identifier: type
}
```

```
fact Membership {
  member: Member
  group: Group
  createdAt: timestamp
}
```

В то время как в `Ownership` родитель является предшественником потомка, в `Membership` группа и член не связаны причинно-следственно. Как показано на рис. 8.5, у них есть общий преемник в членстве.

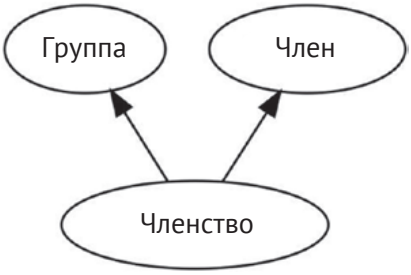


**Рис. 8.5.** Членство является преемником как группы, так и ее члена

Чтобы найти всех членов группы, выполните запрос к ее членству:

```
query membersOfGroup(g: Group) {
  match ms: Membership where ms.group = g
       such that not exists msd: MembershipDeletion where msd.membership
= ms
  then m: Member where m = ms.member
}
```

## Пример

Со временем сотрудники могут быть переведены в разные отделы. Представление должности как отдельного факта, а не как отношения прямого предшественника между отделом и сотрудником, позволяет перевести сотрудника без изменения его идентичности. На рис. 8.6 показано это отношение.

При запросе сотрудников в отделе убедитесь, что в список включены только те назначения, которые не были удалены:

```
query employeesOfDepartment(d: Department) {
  match a: Assignment where a.department = d
       such that not exists ad: AssignmentDeletion where ad.assignment = a
  then e: Employee where e = a.employee
}
```


**Рис. 8.6.** Сотрудник назначается в отдел и впоследствии удаляется из него

## Последствия

Модель не может обеспечить соблюдение бизнес-правила, согласно которому объект должен принадлежать только к одной группе. Не существует механизма, который предотвращает, чтобы два факта членства имели одного и того же предшественника. В реляционной модели ограничение уникальности *могло* бы обеспечить выполнение этого требования. Но ограничение уникальности на одном узле не предотвращает вставку в другой. Ограничение уникальности не может быть выполнено в конечном итоге последовательным образом на нескольких узлах.

Создание и добавление в группу не являются атомарными процессами. В шаблоне *Ownership* родитель создается раньше, чем сущность. В *Membership*, однако, членство создается *после* сущности. Если единственные запросы, которые достигают сущности, проходят через членство, то это не имеет особого значения. Однако если есть другой запрос, который обращается к сущности, то он может быть «*сиротой*» в течение неопределенного периода времени. Если разработчик приложения хочет скрыть «сирот», ему следует добавить в запрос предложение `exists`.

Например, если «Сотрудник» (Employee), определенный ранее, имеет владельца компании как предшественника, то следующий запрос будет включать сотрудников, не назначенных в отдел:

```
query allEmployeesOfCompany(c: Company) {
  match e: employee where e.company = c
}
```

Чтобы исключить неназначенных сотрудников из результатов, разработчик добавляет предложение `exists`, требующее, чтобы назначение было сделано и впоследствии не было удалено:

```
query allEmployeesOfCompany(c: Company) {
  match e: employee where e.company = c
       such that exists a: Assignment where a.employee = e
              such that not exists ad: AssignmentDeletion where ad.assignment
       = a
}
```

### Связанные шаблоны

Если модель требует, чтобы сущность была членом только одной группы, и эта группа не может меняться, то следует использовать шаблон *Ownership*.

Если модель требует, чтобы членство в одной группе было заменено на членство в другой группе, то рассмотрите возможность применения шаблона *Entity Reference*. Модель членства в группе как ссылка на факт группы заменяет предыдущие ссылки для той же сущности. Хотя это и не предотвратит параллельные изменения, но, по крайней мере, сделает удаление из одной группы и добавление в другую атомарной операцией.

## Изменяемое свойство

**Значение** Представляет значения, которые меняются.

Исторические факты неизменны. Они не изменяются. Тем не менее пользователи ожидают, что смогут изменять свойства. Шаблон «Изменяемое свойство» (*Mutable Property*) представляет изменения свойств с течением времени, используя только неизменяемые факты.

В распределенной системе желательно, чтобы узлы могли действовать изолированно. Пользователь должен иметь возможность изменить свойство, не требуя соединения с другим узлом. Пользователь может находиться на мобильном телефоне, который временно отключен от сервера. Или у него может быть медленное сетевое соединение, и задержка при выполнении подключенного обновления была бы нежелательной.

С возможностью автономного изменения возникает вероятность конфликтов. Отключенный пользователь может изменить то же свойство, что и тот, кто подключен. Или два пользователя на медленном соединении могут изменить одно и то же свойство в более или менее одинаковое время. Когда каждое из их изменений распространится на другое, конфликт будет обнаружен. Система должна предусматривать возможность разрешения этих конфликтов.

### Структура

Изменяемое свойство представляется как факт, имеющий сущность в качестве предшественника и значение в качестве поля. Чтобы отслеживать изменения во времени, он записывает предыдущие версии в набор предшественников.

По соглашению, имя факта добавляет имя свойства к имени сущности. Набор предшествующих версий условно называется `prior`. Этот набор пуст для начального значения.

```
fact Entity {
  identifier: type
}
fact EntityProperty {
  entity: Entity
  value: type
  prior: EntityProperty*
}
```

По мере того как пользователь изменяет свойство, набор `prior` захватывает только самую последнюю версию. При обычных обстоятельствах это формирует линейную цепочку фактов свойств, как показано на рис. 8.7.

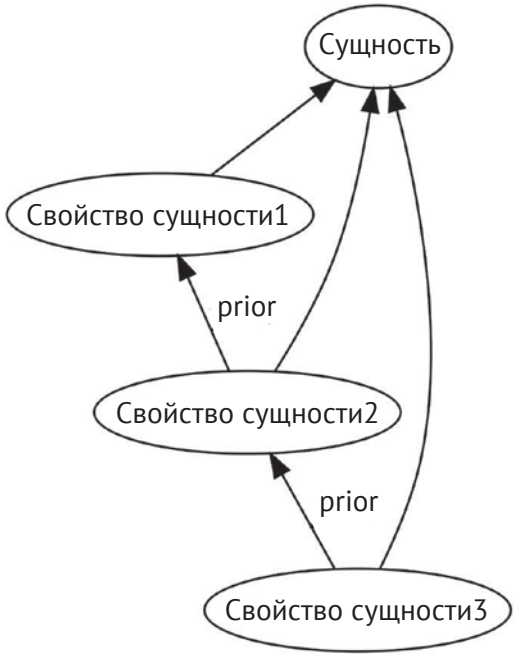


**Рис. 8.7.** В цепочке версий каждая указывает на своего непосредственного предшественника

Если два пользователя (или один пользователь на двух устройствах) параллельно изменяют свойство, граф будет разветвляться. В результате получится дерево, как на рис. 8.8, с более чем одним листом.

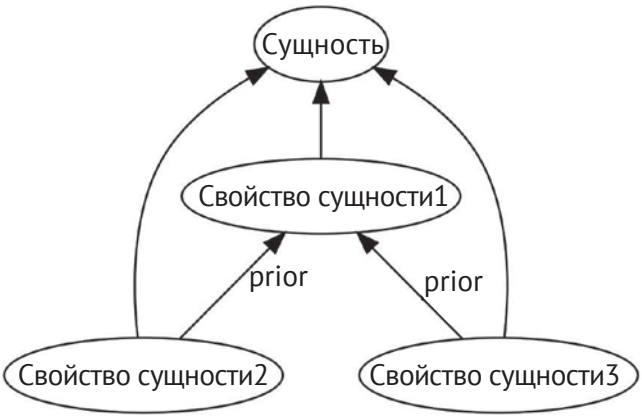


**Рис. 8.8.** Параллельные изменения приводят к появлению нескольких листьев

Когда узел вычисляет дерево с несколькими листьями, он распознает параллельное изменение. В этой ситуации приложение обычно представляет все листья в качестве значений-кандидатов. Каждый лист представляет значение, которое было параллельно установлено для свойства и не было заменено. Пользователь может выбрать одно из значений-кандидатов и разрешить спор.

В качестве альтернативы приложение может самостоятельно вычислить решение. Обычно это выполняется с помощью простой функции над листьями, например максимума. В редких ситуациях, однако, разработчик приложения может выбрать решение на основе ближайшего общего предка всех листьев. Одним из примеров является система контроля версий, такая как Git, которая вычисляет трехстороннее слияние. Такая сложная функция не подходит для большинства приложений.

В любом случае узел определяет, что представить, но он *не генерирует никаких новых фактов*. Факты генерируются только в результате решения пользователя. Когда пользователь изменяет свойство из параллельного состояния, система включает все листья дерева в предварительный набор нового факта. В результате получается граф, как на рис. 8.9, который снова имеет один лист.

Чтобы вычислить набор листьев, узел просто выполняет запрос с предложением `not exists` к предыдущему набору:

```
query valuesOfProperty(e: Entity) {
  match p: EntityProperty where p.entity = e
       such that not exists n: EntityProperty where n.prior = p
}
```

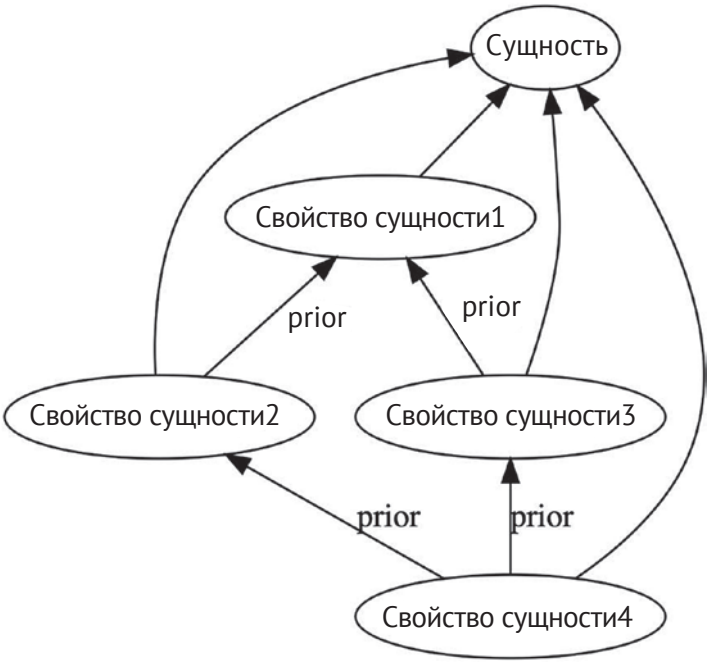


**Рис. 8.9.** Граф снова имеет единственный лист

Если запрос возвращает один факт, то этот факт представляет собой самую последнюю версию. Если он возвращает много фактов, то они представляют собой листья и могут быть использованы в качестве значений-кандидатов.

Свойство не обязательно должно иметь единственное поле значения. Нередко несколько значений изменяются как единое целое. В проектировании, ориентированном на домен (Domain-Driven Design), такая ситуация возникает, когда свойство использует *тип значения*. В таких ситуациях все компоненты типа значения появляются как поля в одном факте свойства.

## Пример

Заказ в нашем примере компании имеет адрес для выставления счета. Это набор полей, которые изменяются как единое целое. Не имеет смысла изменять, например, штат, не изменив также город и улицу. Поэтому поля адреса для выставления счета рассматриваются как единый атомарный факт.

```
fact OrderBillingAddress {
  order: Order
  street: string
  city: string
  state: string
  zipCode: string
  prior: OrderBillingAddress*
}
```

Текущий адрес для выставления счета заказа можно получить с помощью следующего запроса:

```
query billingAddressOfOrder(o: Order) {
  match ba: OrderBillingAddress where ba.order = o
       such that not exists n: OrderBillingAddress where n.prior = ba
}
```

Если запрос возвращает один факт, то он представляет собой самый последний адрес для выставления счетов. Если, однако, возвращается несколько адресов, значит, произошли параллельные изменения, и факты представляют адреса-кандидаты. Приложение представляет все кандидатуры, чтобы пользователь мог изучить и решить проблему.

Заказ также включает адрес доставки. Он представлен как отдельный факт от адреса выставления счета, хотя имеет в основном те же поля.

```
fact OrderShippingAddress {
  order: Order
  street: string
  city: string
  state: string
  zipCode: string
  prior: OrderShippingAddress*
}
```

Аналогичный запрос получает последний адрес доставки. Хотя необычно изменять только одну часть адреса в любой момент времени, нередко изменяется только адрес доставки или только адрес выставления счета. Именно поэтому разработчик приложения решил сделать их отдельными фактами.

В то время как параллельные изменения в адресе выставления счета приведут к появлению нескольких листьев, параллельные изменения между адресом выставления счета, с одной стороны, и адресом доставки – с другой – не приведут. Система просто представит самый последний адрес выставления счета рядом с самым последним адресом доставки. Это отражает намерение разработчика приложения, что адрес доставки и адрес выставления счета не имеют причинно-следственной связи между собой.

## Последствия

Узлы, использующие шаблон *Mutable Property*, могут действовать автономно. Они могут записывать новое значение для свойства без предварительного соединения с любым другим узлом, чтобы предотвратить параллельные изменения.

Иначе говоря, параллельные изменения *невозможно предотвратить*. Не существует механизма в исторической модели, который бы гарантировал, что в любой момент времени может быть сделано только одно изменение. Свойства не могут быть ни заблокированы, ни сериализованы.

Узлы узнают *постфактум*, что параллельные изменения произошли. Все узлы в конечном итоге получат один и тот же граф, вычислят одни и те же листья и, следовательно, придут к одному и тому же выводу. Параллельные изменения не приводят к конфликтам.

Когда пользователь пытается изменить свойство, приложение должно сначала проверить, действительно ли значение было изменено. Приложение может, например, отобразить диалоговое окно с кнопками «ОК» и «Отмена». Пользователь может нажать кнопку «ОК», даже если он не внес никаких изменений. Если бы приложение создало новый факт, не проверив, изменилось ли значение, это создало бы неоправданно сложную историю.

Факт с изменяемым свойством не должен содержать никакой аудиторской информации. Это позволяет двум разным пользователям изменять свойство на одно и то же значение без введения параллельного обновления. Если бы факт содержал, например, пользователя или временную метку, то два параллельных изменения одного и того же значения выглядели бы как разные факты. В результате было бы ненужное слияние между аналогичными изменениями.

Решение по нескольким листьям должно быть основано только на информации, содержащейся в самих фактах. Оно не должно основываться, например, на порядке, в котором факты поступили в узел. Результат – это функция, которая является коммутативной и детерминированной, она вычисляет один и тот же результат в каждом узле, независимо от порядка сообщений. Вот почему результаты запросов являются неупорядоченными *наборами*, а не упорядоченными *списками*.

Если узел вычисляет разрешение параллельного изменения, он должен делать это только при чтении. Он не должен пытаться создать новый факт для обработки параллельных изменений. Сделать это означало бы создать возможность бесконечного шквала параллельных изменений. Алгоритмы консенсуса, такие как Paxos, тщательно разработаны, чтобы избежать этих шквалов, но без детального рассмотрения такой шквал может легко возникнуть. В любом случае, сильная конечная согласованность требует сходимости без консенсуса. Это достижимо, если все узлы выполняют одну и ту же детерминированную функцию при чтении.

Шаблон *Mutable Property* не может гарантировать, что свойство имеет единственное значение. В результате запроса всегда будет получен набор. Приложения должны быть написаны с учетом того, что этот набор может иметь несколько значений. Хотя иногда возникает соблазн ввести правило, специфичное для конкретного местоположения, чтобы предотвратить параллельные обновления (например, только одному пользователю разрешено изменять свойство, или только один узел может быть использован для такого изменения), такие правила в конечном счете трудно обеспечить, и они накладывают нежелательные ограничения на систему.

Запрос на текущее значение свойства может вернуть пустой набор. Это представляет собой ситуацию, когда свойство не было инициализировано. На удаленных узлах это может также означать, что сущность была передана, но ее начальные свойства еще не получены. Создание сущности не является атомарным с инициализацией ее свойств. Если разработчик приложения на-

меревается представить только инициализированные сущности, он может добавить предложение `exists`, основанное на факте свойства.

### Связанные шаблоны

Если изменяемое свойство представляет собой отношение с другой сущностью, то шаблон становится ссылкой на сущность (*Entity Reference*).

## Ссылка на сущность

**Значение** Представляет изменяемые отношения между сущностями.

Там, где *Ownership* (Владение) и *Membership* (Членство) являются строгими конструкциями группировки, некоторые отношения между сущностями являются простыми ссылками. Эти ссылки не подразумевают никакого вида принадлежность или группировку, а скорее просто ассоциацию.

Ссылка на сущность – это свойство, которое указывает на другую сущность. В Domain-Driven Design сущность, на которую ссылаются, обычно является *корнем агрегата*, возможно, в другом *ограниченном контексте*. В объектно-ориентированном языке ссылка на сущность – это указатель на другой объект. А в реляционной базе данных это внешний ключ. Ссылка обычно изменяема и часто инициализируется значением NULL.

В реляционной базе данных внешние ключи используются для представления владения, членства и ссылки на сущность. Чтобы отличить их друг от друга, сначала обратите внимание на кардинальность. Отношение «многие ко многим» обычно обозначает членство. Отношение «один ко многим», которое имеет ограничение каскадного удаления, обозначает владение. Менее ограниченное отношение «один ко многим», особенно допускающее NULL, вероятно, является ссылкой на сущность.

### Структура

Структура ссылки на сущность очень похожа на структуру изменяемого свойства. Это факт, имеющий в качестве предшественников первичную сущность и ссылочную сущность. Ссылочная сущность часто нулевая. Так же, как и изменяемое свойство, факт хранит набор предыдущих версий ссылок на сущность.

```
fact EntityReference {
  entity: Entity
  referencedEntity: ReferencedEntity?
  prior: EntityReference*
}
```

Различие между первичной и ссылочной сущностями является важным. Первичная сущность – это сущность со свойством ссылки. Создание нового факта *EntityReference* изменяет значение этого свойства для первичной сущности. Предыдущий набор будет включать другие факты, которые ссылаются на ту же первичную сущность.

Запрос текущего значения ссылки на сущность начинается с первичной сущности. Как и запрос свойства в шаблоне *Mutable Property*, он соответствует ссылкам, которые не были заменены. Запрос включает одно дополнительное предложение, которое следует за ссылкой на сущность.

```
query referencedEntity(e: Entity) {
  match er: EntityReference where er.entity = e
        such that not exists n: EntityReference where n.prior = er
  then re: ReferencedEntity where re = er.referencedEntity
}
```

В отличие от изменяемых свойств, ссылки на сущность позволяют выполнять запросы в обратном направлении. Чтобы выполнить запрос от ссылочной сущности обратно ко всем сущностям, которые на нее ссылаются, включите предложение `not exists` в `prior`. Это предотвратит возврат запросом сущностей со ссылками, которые были заменены.

```
query entitiesReferencing(re: ReferencedEntity) {
  match er: EntityReference where er.referencedEntity = re
        such that not exists n: EntityReference where n.prior = er
  then e: Entity where e = er.entity
}
```

## Пример

Строка заказа ссылается на продукт, который был оплачен. Эта связь является необязательной: некоторые строки заказа представляют сборы, скидки или услуги, не указанные в каталоге. Поэтому факт «Строка заказа» (OrderLine) имеет ссылку на сущность «Продукт» (Product).

```
fact OrderLineProduct {
  orderLine: OrderLine
  product: Product?
  prior: OrderLineProduct*
}
```

Это создает отношение, показанное на рис. 8.10.

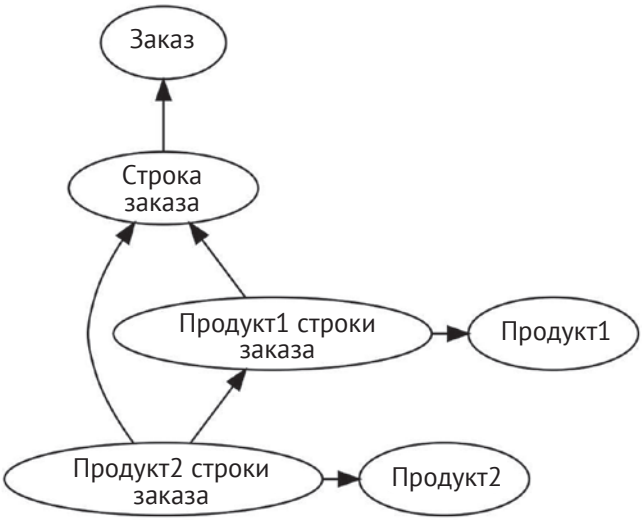


**Рис. 8.10.** Две версии строки заказа, каждая из которых ссылается на разный продукт

Запрос на продукт, на который ссылается строка заказа, начинается, как любой другой запрос на изменяемое свойство. Но затем он содержит дополнительное предложение для получения ссылочного продукта.

```
query productForOrderLine(ol: OrderLine) {
  match olp: OrderLineProduct where olp.orderLine = ol
       such that not exists n: OrderLineProduct where n.prior = olp
  then p: Product where p = olp.product
}
```

Обходя граф в обратном направлении, мы можем запросить заказы, которые покупают данный продукт. Этот запрос включает то же предложение `not exists`.

```
query ordersContainingProduct(p: Product) {
  match olp: OrderLineProduct where olp.product = p
       such that not exists n: OrderLineProduct where n.prior = olp
  then o: Order where o = olp.orderLine.order
}
```

Сходство между двумя запросами заставляет их вести себя атомарно. Когда строка заказа меняется на ссылку на другой продукт, это затрагивает оба запроса. Первый запрос больше не будет возвращать ранее упомянутый продукт, а второй запрос не будет возвращать заказ.

## Последствия

Как и в случае с шаблоном *Mutable Property*, *Entity Reference* не может гарантировать, что он ссылается только на одну сущность. Запрос для текущей ссылки

возвращает набор. Приложение должно соответствующим образом реагировать на набор, превышающий 1. Это представляет собой параллельное обновление ссылок на сущность.

Результатом запроса может быть и пустой набор. Это может произойти, как и в случае с *Mutable Property*, когда ссылка еще не инициализирована. Но это также может произойти, когда ссылка была установлена в NULL.

### Связанные шаблоны

Это вариант шаблона *Mutable Property*, в котором значением свойства является ссылка на другую сущность.

Иногда этот шаблон используется как альтернатива шаблону *Membership*, чтобы указать, что сущность должна быть членом только одной группы. Хотя он не может обеспечить соблюдение этого правила, он, по крайней мере, делает переход из одной группы в другую атомарной операцией.

## Шаблоны рабочих процессов

Хотя операции CRUD составляют важную часть разработки бизнес-приложений, это еще не вся история. Следующий набор бизнес-операций, который необходимо рассмотреть, связан с прохождением сущностей через рабочий процесс. Рабочий процесс обычно относится к области моделирования бизнес-процессов, диаграмм перехода состояний и блок-схем. Это изучение совместных шагов, которые перемещают работу от начала до завершения.

Отслеживание потока работ в системе, допускающей изменения, – это упражнение в разочаровании. Если каждый шаг процесса потенциально может изменить элемент работы, то рассуждения о поведении системы требуют тщательного анализа всех возможных перестановок. Позволяя параллельное выполнение, изменяемость часто приводит к состоянию гонки[1].

Но в рамках неизменяемой архитектуры рабочий процесс намного проще. Он начинается с фиксации работы, которая должна быть выполнена в неизменяемом объекте. Любые дальнейшие изменения в исходном объекте игнорируются. Затем мы создаем запрос, чтобы определить, какие рабочие элементы готовы для любого заданного процесса. Наконец, мы фиксируем результат этого процесса неизменяемым образом, который атомарно перемещает рабочий элемент на следующий шаг.

Добавление рабочего процесса в приложение превращает его из анемичной модели форм над данными в систему, которая может помочь в общении между сотрудниками. Не каждая подобласть приложения нуждается в рабочем процессе, но центральные ограниченные контексты, которые предоставляют наибольшую ценность для бизнеса, часто необходимы.

[1] Состояние гонки – ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. – *Прим. перев.*

## Транзакция

**Значение** Захват известного состояния сущности для выполнения атомарной единицы работы.

Структурные шаблоны, которые мы только что рассмотрели, позволяют сущности изменяться во времени. Изменения фиксируются как неизменные факты, но накопление новых фактов по мере взаимодействия пользователя с системой имитирует изменения сущности. В какой-то момент пользователь решит предпринять какое-то действие. Любые дальнейшие изменения сущности после этого момента не должны влиять на это действие.

Пользователи могут добавлять товары в корзину. Они могут удалять товары, заменять их и возвращать обратно в корзину. Они могут изменять количество, товар, варианты доставки, адрес доставки и любое другое свойство. Структурные шаблоны, описанные в предыдущем разделе, позволяют выполнять эти операции.

Но затем, когда пользователь отправляет заказ, элементы и все их свойства должны быть зафиксированы. Никакие дополнительные элементы не могут быть добавлены, и никакие свойства не могут быть изменены. Обработка может начаться в любое время, и изменение заказа в процессе выполнения будет нарушением для бизнеса.

Шаблон *Transaction* («Транзакция») использует преимущества неизменяемости для обработки бизнес-данных. Он записывает информацию о запросе на выполнение работы таким образом, что она не может быть изменена после начала работы.

### Структура

Транзакция идентифицирует в качестве предшественника сущность, на которую она действует. В то время как эта сущность изначально была отправной точкой для дочерних элементов, изменяемых свойств и других преемников, теперь транзакция стремится зафиксировать ее. Она делает это, инвертируя отношение предшественник/преемник.

Если *Ownership* помещает родителя в качестве предшественника его детей, то *Transaction* делает детей предшественниками родителей. Преемники могут быть добавлены со временем, но предшественники неизменны. Запись детей в качестве предшественников предотвращает дальнейшее создание или удаление.

Транзакция также определяет конкретные версии изменяемых свойств (*Mutable Properties*). Они становятся прямыми или косвенными предшественниками транзакции. Опять же, отношение инвертируется таким образом, что любые дальнейшие модификации этих свойств не влияют на транзакцию.

Элемент транзакции (TransactionItem), как описано ниже, является дочерним элементом транзакции. Транзакция имеет набор предшественников TransactionItem. Кроме того, элемент транзакции фиксирует одну конкретную версию изменяемого свойства.

```
fact TransactionItem {
  itemContext: ChildEntity
  property: ChildProperty
}
fact Transaction {
  transactionContext: ParentEntity
  items: TransactionItem*
}
```

Транзакция и все ее элементы фиксируются в одном компьютере. Как правило, это рабочая станция, которую использует лицо, принимающее решение. Когда пользователь принимает решение о выдаче транзакции, система фиксирует состояние объектов в том виде, в котором они известны пользователю в данный момент. Создание транзакции не требует связи с каким-либо другим узлом, поскольку вся необходимая информация является локальной.

## Пример

Когда клиент отправляет заказ, он фиксирует его текущее состояние. Он не может вносить дальнейшие изменения в заказ. Он может только запрашивать последующий возврат и новые заказы.

Мы начнем с существующей структуры заказа, используя шаблоны *Entity*, *Delete* и *Mutable Property*. Затем создадим параллельную модель, как показано на рис. 8.11, которая инвертирует отношения предшественник/преемник. Элементы в заказе становятся предшественниками, так чтобы новые элементы не могли быть добавлены.


**Рис. 8.11.** Отправка заказа инвертирует модель, чтобы зафиксировать предшественников

Все удаленные строки не включаются в «Передачу заказа» (OrderSubmission). Другие строки могут быть впоследствии удалены, или удаленные строки могут быть восстановлены. Ни одно из этих изменений не повлияет на зафиксированные строки переданного заказа (OrderSubmissionLines).

Все стрелки направлены от OrderSubmission. Всю информацию, необходимую для обработки заказа, можно найти, пройдя по графу в одном направлении. Имея OrderSubmission, любой узел вычислит точно такой же заказ. Это фиксирует элементы, продукты и количества.

### Последствия

После записи транзакции последующие изменения сущностей или свойств не будут иметь никакого эффекта. Вся информация в транзакции записывается в отношениях предшественников. Предшественники неизменяемы, поэтому транзакция блокируется.

Все узлы, принимающие транзакцию, видят ее в абсолютно одинаковом состоянии. Идентичность факта включает в себя идентичность его предшественников. Любое различие в предшественниках, таких как элементы транзакции или версии свойств, обязательно приведет к разным фактам.

Транзакция обрабатывается атомарно. Элементы могут прибыть в узел раньше, чем их транзакции. Но обработка начинается с запроса транзакции, а не элемента. Элементы будут «спящими» до поступления последующей транзакции, в это время все элементы вступят в силу одновременно.

Вся необходимая информация должна находиться в транзитивном замыкании транзакции. Начиная с факта транзакции, проследите всех предшественников. От этих фактов рекурсивно перейдите к их предшественникам. Транзитивное замыкание – это набор всех посещенных таким образом фактов.

### Связанные шаблоны

Шаблон *Transaction* (Транзакция) инвертирует отношение предшественник/преемник в шаблонах *Ownerhship* (Владение) и *Mutable Property* (Изменяемое свойство).

Транзакция часто помещается в *Queue* (Очередь) или в папку *Outbox* (Исходящие) и обычно ассоциируется с шаблоном *Period* (Период).

## Очередь

**Значение** Управление работой, которую необходимо обработать вручную.

Работа, которую необходимо обработать, часто представлена в виде списка. Пользовательский интерфейс показывает пользователю набор рабочих элементов, требующих его внимания. Пользователь выбирает рабочий элемент и переходит в ту часть приложения, где он может его обработать.

Пользователь может прервать работу. Поэтому работа остается в очереди до тех пор, пока пользователь не завершит ее. Если другой пользователь наблюдает за очередью, он сможет увидеть тот же самый рабочий элемент.

Шаблон *Queue* (Очередь) представляет набор рабочих элементов, готовых к ручной обработке. Он гарантирует, что рабочий элемент будет удален из очереди, когда его обработка будет завершена.

## Структура

Очередь – это не что иное, как запрос, который возвращает факты, представляющие работу, которая должна быть выполнена. Запрос начинается с некоторой сущности корневого уровня, например «Владельца» (Owner), и ищет дочерние сущности, для которых не было выполнено какое-либо действие.

```
query workToDo(o: Owner) {
  match wi: WorkItem where wi.owner = o
       such that not exists a: Action where a.workItem = wi
}
```

В процессе выполнения запрошенной работы пользователь создаст факт «Действие» (Action). Действие записывает результат работы.

Поскольку действие появляется в предложении `not exists`, рабочий элемент удаляется из запроса после выполнения действия. Запись действия и удаление работы из очереди происходят в рамках одной атомарной операции.

## Пример

После поступления заказа в нашу гипотетическую компанию отдел доставки отбирает товар для выполнения заказа и печатает упаковочный лист. Отдел логистики тем временем вызывает грузовик для доставки. Каждый из этих процессов выполняется вручную. Взаимосвязь между подачей заказа, доставкой и упаковочными листами отражена в модели, изображенной на рис. 8.12.

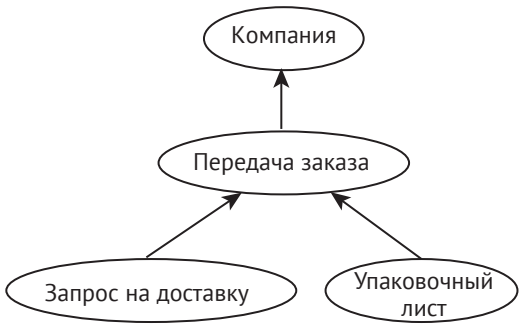


**Рис. 8.12.** Подача заказа вызывает как запрос на доставку, так и упаковочный лист

Менеджер по доставке знает, какие заказы нужно собрать на основе запроса. Запрос ищет заказы, которые еще не имеют упаковочного листа.

```
query ordersToPick(c: Company) {
  match o: OrderSubmission where o.company = c
       such that not exists ps: PackingSlip where ps.orderSubmission = o
}
```

После того как заказ собран, менеджер по доставке печатает упаковочный лист. Это действие удаляет заказ из очереди.

Тем временем команда логистики выполняет еще один запрос, чтобы найти заказы, которые еще не имеют запроса на доставку:

```
query ordersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
   such that not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

Они вызывают грузовик, а затем вводят в систему запрос на доставку (RequestForDelivery). Как только они делают это, заказ больше не появляется в запросе. Он был удален из очереди.

Мы намеренно решили не устанавливать отношения предшественник/преемник между запросом на доставку и упаковочным листом. Запрос на доставку может быть сделан до того, как товар будет собран. Или, в зависимости от объема, склад может оказаться переполненным, и запрос на доставку будет отложен.

Когда менеджер по отгрузке прогнозирует, что склад скоро будет переполнен, он уведомляет логистов о необходимости переключиться на другой запрос. Теперь они ждут, пока заказы не будут собраны перед запросом на доставку.

```
query pickedOrdersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
     such that exists ps: PackingSlip where ps.orderSubmission = o
     and not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

Создание упаковочного листа атомарно перемещает работу из очереди менеджера по доставке в очередь логистики. Последующее создание запроса на доставку удаляет его из очереди логистики.

Упаковочный лист не является жестким предварительным условием. Он не является предшественником запроса на доставку. Но, переключаясь с одной очереди на другую, компания может корректировать свой бизнес-процесс, чтобы лучше реагировать на обстоятельства.

## Последствия

Действие, выполненное над рабочим элементом, используется в предложении `not exist` очереди. В результате запись действия и удаление рабочего элемента из очереди являются одной атомарной операцией.

В отличие от очереди FIFO (первым вошел – первым вышел), запрос очереди не навязывает порядок рабочих элементов. Результаты запроса представляют собой набор, а не список. Если порядок важен, поставьте метку времени на факте рабочего элемента. Используйте эту метку, чтобы упорядочить набор для представления пользователю.

### Связанные шаблоны

Если работа должна выполняться автоматически, а не вручную, то больше подходит шаблон «Исходящие» (*Outbox*).

Рабочие элементы в очереди часто являются транзакциями (*Transaction*s).

Рабочие элементы нередко группируются по единицам времени. Это отражает естественный период, который уже признан бизнесом. Применение шаблона «Период» (*Period*) имеет дополнительное преимущество в том, что оно предотвращает замедление выполнения запроса очереди по мере накопления истории.

## Период

**Значение** Связывает накопление фактов с дискретными временными интервалами.

Время, необходимое для запроса исторической модели, зависит от количества преемников у начальной точки или промежуточного факта. Если мы начнем каждый запрос с корня графа, эти запросы будут становиться медленнее с течением времени. Если начинать дальше по графу с факта, который имеет ограниченное число преемников, производительность будет оставаться постоянной по мере накопления количества фактов.

Любая функция системы, ограничивающая число преемников, является хорошим кандидатом для деления графа на части. Наиболее доступной характеристикой является время. Шаблон «Период» (*Period*) подразделяет исторический граф на дискретные единицы времени. Если общее количество фактов будет расти, количество фактов *на единицу периода* будет оставаться несколько более ограниченным.

В дополнение к преимуществам производительности ассоциирование фактов с периодом часто отражает важную бизнес-концепцию. Бухгалтеры, как правило, закрывают свои книги по ежедневным, месячным и квартальным периодам. Они делают это не только для того, чтобы ограничить размер бухгалтерской книги, но и для того, чтобы установить для себя границы отчетности. Шаблон *Period* стремится сделать то же самое с прикладными данными.

### Структура

Период – это факт, который имеет одного предшественника – собственника (*Owner*), придающего ему контекст, и одно дискретное значение времени. Время измеряется в грубых единицах; это не непрерывная временная метка.

```
fact Period {
  owner: Owner
  time: discreteTime
}
```

Типичными вариантами дискретной единицы времени являются календарный или рабочий день, месяц, квартал или год. Для систем с высокой пропускной способностью единицы могут быть вплоть до часа, но редко меньше.

Рабочие элементы включают период в качестве предшественника. Запросы на работу начинаются с периода.

```
query workToDo(p: Period) {
  match wi: WorkItem where wi.period = p
       such that not exists a: Action where a.workItem = wi
}
```

Результаты двух или более запросов объединяются вместе, чтобы создать перекрывающийся запрос. Перекрытие выбирается таким образом, чтобы у удаленных узлов было достаточно времени для подключения и обмена своими рабочими элементами и чтобы эти рабочие элементы были обработаны до истечения периода.

Период не имеет дополнительных полей, так что владелец и дискретная единица времени создают уникальный факт. Все узлы, создающие рабочие элементы, производят один и тот же факт. И каждый запрос на рабочие элементы создает одну и ту же начальную точку.

Иногда периоды организованы иерархически. Самый большой период, например год, попадает непосредственно под владельца. Следующий ниже период, например квартал, имеет более крупный период как предшественника. Периоды, организованные таким образом, должны иметь общую границу, месяц и неделя не могут быть организованы в иерархию. Обычно это делается для отчетности, а не по соображениям производительности.

## Пример

В предыдущем примере мы добавили отправку заказов непосредственно компании. Со временем система ищет больше заказов компании, чтобы найти те, которые еще не были отобраны или отправлены. Мы можем облегчить работу системы и записать важное свойство модели.

Факт DateOfBusiness имеет одного предшественника и одно поле:

```
fact DateOfBusiness {
  company: Company
  businessDate: date
}
```

Мы вставляем DateOfBusiness между компанией и передачей заказа, как показано на рис. 8.13. Этот факт фиксирует дату, когда заказ был представлен.

Дата бизнеса не определяется строго по компьютерным часам. Заказ может быть рассчитан в следующую дату бизнеса, если он был размещен в нерабочее время, или если это произошло в выходной или праздничный день. Фактически компания может даже выбрать политику, при которой заказы, размещенные после 3:00, относятся к следующей дате бизнеса. Период – это операционная, а не физическая конструкция.


**Рис. 8.13.** Промежуточный факт группирует заказы, поступившие в компанию, по дате

Не все узлы должны переходить к следующему периоду в одно и то же время. Нет необходимости в строгой синхронизации часов на рабочих станциях, на которых пользователи подают заявки. Если одна рабочая станция начинает подавать заказы на следующую дату бизнеса, в то время как другая рабочая станция остается на текущей, то эти заказы просто считаются в разных периодах. Это не вызовет никаких проблем до тех пор, пока нет причинно-следственной связи между подачей заказов. А тот факт, что разработчик решил не делать одну подачу заказа предшественницей другой, указывает на то, что причинно-следственной связи быть *не должно*.

Чтобы определить заказы, которые необходимо отобрать для любой конкретной даты бизнеса, мы находим те, которые еще не имеют упаковочного листа:

```
query ordersToPick(dob: DateOfBusiness) {
  match o: OrderSubmission where o.dateOfBusiness = dob
       such that not exists ps: PackingSlip where ps.order = o
}
```

Мы выполняем этот запрос для двух дат бизнеса – предыдущей и текущей – и объединяем наборы. Любое данное представление заказа встречается только в одной дате бизнеса, на что указывает единственность его предшественника, поэтому такая практика не грозит дублированием. Однако перекрытие *не позволяет* нам пропустить заказы. До тех пор, пока период значительно больше, чем SLA[1], мы получим и обработаем заказы до того, как перенесем запрос слишком далеко вперед.

[1] Соглашение об уровне обслуживания (Service Level Agreement, SLA) – это договор между заказчиком услуги и ее исполнителем. Основные пункты SLA: описание обязательств сторон; указание сроков действия соглашения; описание процесса использования услуг; описание процедуры контроля за исполнением соглашения; описание процедуры восстановления работы в случае перебоев и связанных с ними штрафных санкций; процедуры решения технических проблем и спорных вопросов. – *Прим. перев.*

### Последствия

У элемента работы должен быть только один связанный период. Если единица работы разбита на меньшие единицы и каждая из них имеет свой собственный период, то можно разделить работу между двумя периодами. Представьте себе поезд, движущийся через стрелку в момент, когда она переключается. Если вагоны не соединены, то проблемы нет. Но если они прицеплены друг к другу, это может привести к нежелательным последствиям.

Для поиска рабочих элементов следует запрашивать два или более периодов. Перекрытие обеспечивает буфер времени для поступления и обработки рабочих элементов. Если вышестоящие узлы могут быть отключены, количество и продолжительность перекрывающихся периодов должны быть выбраны таким образом, чтобы они могли повторно подключиться. До тех пор, пока ожидаемое время получения и обработки рабочих элементов значительно меньше, чем продолжительность перекрывающихся периодов, работа, как правило, не будет потеряна.

Однако в модели нет механизма, гарантирующего, что работа не будет отложена за пределы самого раннего запрашиваемого периода. Система обработки должна быть достаточно гибкой, чтобы ее можно было вручную перенастроить для подбора отстающих элементов работы. Запрос от владельца на одного предшественника выше, чем период, может охватывать все периоды. Хотя этот запрос будет более медленным, он позволит заглянуть в прошлое, чтобы найти все пропущенные элементы работы. Затем может быть принято бизнес-решение для определения наилучшего корректирующего действия.

### Связанные шаблоны

Периоды часто используются в качестве отправной точки для запросов в шаблоне «Очередь» (Queue) или «Исходящие» (Outbox). Неограниченная очередь постепенно становится проблемой производительности. Но очередь, ограниченная ожидаемым количеством рабочих элементов за период, намного проще в управлении.

Рабочие элементы внутри периода часто являются транзакциями (Transactions).

## Исходящие

**Значение** Отправляет работу во внешнюю систему, которая не следует принципам неизменяемой архитектуры.

Распределенные системы неоднородны. Компоненты, разработанные с различными архитектурными ограничениями, должны будут взаимодействовать друг с другом. Мы можем столкнуться с необходимостью посылать запросы из неизменяемой системы в систему, зависящую от местоположения.

На границе между неизменяемой и зависящей от местоположения системами у нас есть API.

Неизменяемая система запускает сервис, который вызывает API всякий раз, когда факт появляется в очереди (*Queue*). Затем она записывает результаты этого вызова API в новый факт, который удаляет работу из очереди.

Один экземпляр сервиса было бы легко реализовать, но он не обеспечит ни высокой доступности, ни высокой пропускной способности. Для обеспечения этих свойств нам нужна избыточность. И вот тут-то реализация сервиса становится сложной.

При отправке работы в API, зависящий от местоположения, часто полезно ограничить количество дублирующих запросов. Если система не является идемпотентной, она может ошибочно дублировать работу. В этом случае мы хотели бы гарантировать, насколько это возможно, чтобы запросы отправлялись ровно один раз. Но даже если нисходящая система является идемпотентной, умножение каждого запроса на количество параллельных сервисов является неоправданно расточительным.

Шаблон «Исходящие» (*Outbox*) предоставляет механизм, с помощью которого параллельные сервисы избегают отправки дублирующихся рабочих запросов в сторонние системы. Он не может полностью предотвратить дублирований, но может предпринять шаги для их уменьшения.

Обратите внимание, что соответствующего шаблона «Входящие» не существует. Когда информация поступает из внешней системы, она просто превращается в факт. Никакого специального шаблона преобразования в этом направлении не требуется.

## Структура

Шаблон *Outbox* интегрируется с сервисами, зависящими от местоположения, сам становясь зависимым от местоположения. В отличие от других представленных здесь шаблонов, этот не реализован полностью в рамках правил неизменяемой архитектуры. Вместо этого он использует специфичный для конкретного местоположения журнал для отслеживания успешных вызовов API.

### Журналирование

В журнал записываются результаты запросов API, сделанных к удаленной системе. Индексом в журнале является хеш факта рабочего элемента, который запустил вызов API. Журнал содержит все соответствующие данные, полученные от API. В нем записываются только успешные вызовы API.

Если все работает правильно, сервис выполняет следующие действия по порядку:

1. Получает факт рабочего элемента из запроса очереди.
2. Вызывает API.
3. Сохраняет результаты вызова API в журнале.
4. Создает факт с результатами вызова API.

Факт, созданный на шаге 4, также имеет эффект удаления рабочего элемента из очереди. В следующий раз, когда сервис выполнит запрос, факт рабочего элемента не будет присутствовать. Это и есть «успешный путь».

Когда что-то работает некорректно, сервис может потерпеть неудачу на части пути и повторить эти шаги. Журнал предназначен для уменьшения вероят-

ности того, что API будет вызываться более одного раза. Для этого он предоставляет возможность пропустить вызов API на шаге 2 в некоторых сценариях отказа.

После того как сервис получает факт (шаг 1), он проверяет журнал на наличие совпадающей строки. Журнал индексируется по хешу факта рабочего элемента. Если факт найден, значит, предыдущий или параллельный вызов сервиса выполнил шаг 3. Сервис считывает всю информацию о результате вызова API и переходит к шагу 4 для создания факта.

После того как сервис выполнил вызов API и получил успешный результат, он пытается вставить эту информацию в журнал. Однако журнал имеет ограничение на уникальность хеша факта рабочего элемента. Поэтому вставка будет неудачной, если параллельный сервис вставит свои результаты первым. Когда это происходит, сервис только что обнаружил дублирующий вызов API. Он прерывается и позволяет параллельному вызову завершить работу. Полный поток алгоритма журналирования показан на рис. 8.14.

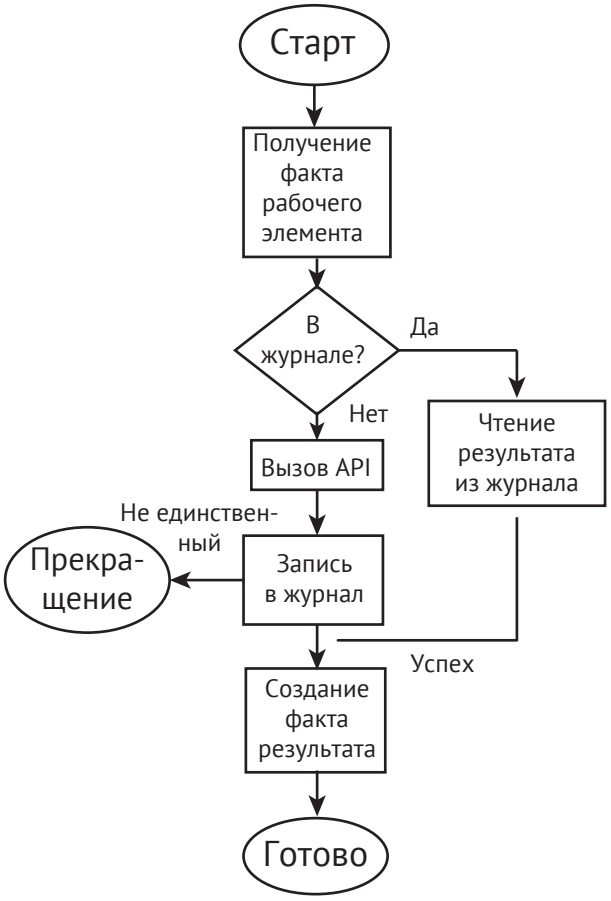


**Рис. 8.14.** Журналирование снижает вероятность дублирования вызовов API

### Случайные задержки обработки

Журналирование снижает вероятность дублирования успешных вызовов API, но не предотвращает их. Один из путей, которым дублирование все же может произойти, заключается в том, что два узла запускают сервис на одном и том же рабочем элементе параллельно. Мы можем предпринять дополнительные шаги, чтобы сделать параллельную работу менее вероятной.

Самый простой способ уменьшить вероятность параллельного выполнения – ввести случайную задержку обработки. Рассмотрим сервис, который использует опрос для запроса очереди рабочих элементов. Он запускается через регулярные промежутки времени и выполняет свой запрос. Если он находит несколько рабочих элементов, то обрабатывает *один из них* и снова выполняет запрос. Он не обрабатывает все элементы, потому что это увеличивает время, в течение которого другой сервис может выполнить тот же запрос. Он просто выбирает случайным образом один рабочий элемент и оставляет остальные в очереди.

Мы можем настроить все узлы на пробуждение сервиса по одному и тому же расписанию. Возможно, мы просто создадим задание, которое будет выполняться раз в минуту. Но, чтобы уменьшить вероятность параллельного выполнения, мы ждем случайное количество секунд перед запуском запроса.

Это очень простая процедура. В сочетании с журналом и относительно быстрым API она может быть довольно эффективной. Но она подходит только для низкоскоростных интерфейсов. Она вносит ненужную задержку и ограничивает частоту, с которой рабочие элементы могут быть обработаны.

### Хеширование рандеву

Когда требуется низкая задержка и высокая пропускная способность, можно использовать более сложный механизм. Хеширование рандеву (Rendezvous Hashing[1]) – это техника уникального распределения объектов узлам. Она часто используется в распределенных кешах. Мы адаптируем ее для того, чтобы распределять факты рабочих элементов сервисам. Аналогичный алгоритм – последовательное хеширование (Consistent Hashing[2]) – может быть адаптирован так же хорошо.

Для начала каждый экземпляр сервиса при запуске генерирует случайное число. Он регистрирует свой номер с другими сервисами. Реестр сервисов может быть реализован с помощью протокола сплетен, распределенного хеша или даже той же базы данных, которая используется для ведения журнала. Единственное требование заключается в том, чтобы другие сервисы узнали о нем вскоре после того, как сервис перейдет в онлайн.

После регистрации сервис подписывается на новые рабочие элементы, поступающие в очередь. В отличие от решения со случайной задержкой обработки,

---

[1] *Thaler David, Chinya Ravishankar*. A Name-Based Mapping Scheme for Rendezvous. University of Michigan Technical Report CSE-TR-316-96.

[2] *Karger D., Lehman E., Leighton T., Panigrahy R., Levine M., Lewin D.* (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing. ACM Press New York, NY, USA. p. 654–663.

сервисы не опрашивают. Они получают уведомления через веб-перехватчики, широковещательную рассылку или очередь сообщений «публикация–подписка», как только работа становится доступной. Какой механизм они используют, зависит от выбранной вами коммуникационной инфраструктуры.

Когда сервис получает рабочий элемент, он вычисляет хеш. Но прежде чем проверить журнал, он сравнивает хеш факта с каждым из случайных чисел всех зарегистрированных сервисов, включая себя. Он вычисляет хеш каждого сравнения, получая *вес*. Сервис с наибольшим весом является тем сервисом, который должен обработать рабочий элемент. Если победителем является сам сервис, то он проверяет журнал и обрабатывает рабочий элемент. Алгоритм показан на рис. 8.15.

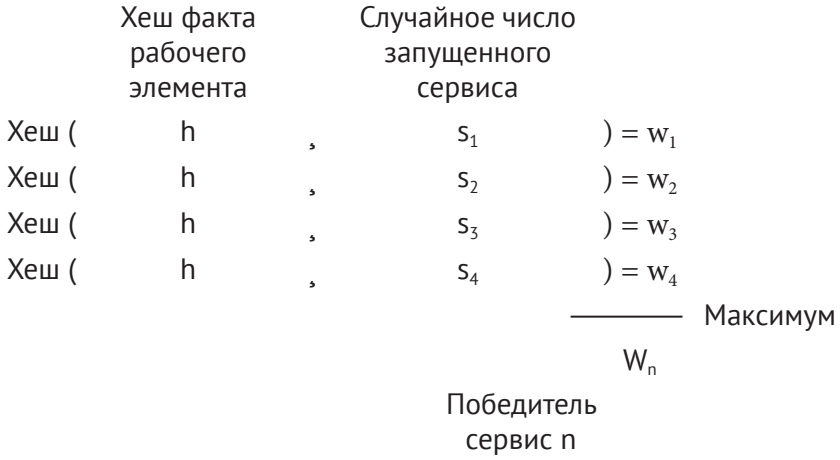


**Рис. 8.15.** Узел вычисляет веса для одного рабочего элемента, чтобы определить выигравший сервис

Все сервисы рассчитают одинаковые веса для рабочего элемента. Поэтому все они выберут одного и того же победителя. Только победитель обработает рабочий элемент, что приведет к уменьшению вероятности параллельного процесса.

**Отказ сервиса**

К сожалению, узлы выходят из строя. Когда сервис перестает отвечать, его рабочие элементы остаются в очереди дольше, чем ожидалось. К счастью, другие сервисы могут это обнаружить.

Поскольку все сервисы вычисляют веса для всех рабочих элементов, каждый сервис может увидеть свое место. Если сервис определяет, что занимает второе место, то он продолжает следить за рабочим элементом. Если после тайм-аута он снова видит это, то считает, что занявший первое место сервис отказал. Он обрабатывает рабочий элемент и удаляет отказавший узел из реестра.

Если сервис, который *не* отказал, обнаруживает, что удален из реестра, он просто создает новое случайное число и возвращается обратно. Тайм-аут дол-

жен быть достаточно длительным, чтобы сделать этот сценарий маловероятным, но достаточно коротким, чтобы отказы не оставались необнаруженными слишком долго.

Обнаружение сбоев может быть обобщено не только для второго места. Третьи, четвертые и более высокие призеры могут устанавливать более длительные тайм-ауты для рабочих элементов. Это позволит смягчить одновременный отказ нескольких сервисов, что может быть вызвано перебоем в работе инфраструктуры или сети.

## Пример

Мы интегрируем нашу компанию из примера со сторонней системой дебиторской задолженности. Когда заказ отправлен, мы передаем его для выставления счета. Наш первый шаг – определить очередь заказов для выставления счетов.

```
query ordersToInvoice(dob: DateOfBusiness) {
  match o: OrderSubmission where o.dateOfBusiness = dob
       such that not exists i: Invoice where i.orderSubmission = o
}
```

После этого мы создадим сервис, который будет подписан на эту очередь. Когда сервис запускается, он генерирует случайное число и вставляет запись в общий кеш Redis. Когда создается новая «Передача заказа» (OrderSubmission), узел, который ее создал, отправляет уведомление. Сервис подписывается на это уведомление, чтобы узнать о новых рабочих элементах.

Получив уведомление, сервис выполняет запрос для поиска рабочих элементов. Он сопоставляет хеш каждого рабочего элемента со случайным номером каждого сервиса в кеше Redis. Он хеширует эту пару, чтобы вычислить вес этого рабочего элемента для данного сервиса. Все рабочие элементы, для которых сам сервис имеет наибольший вес, переходят к следующему шагу.

Сервис проверяет общую базу данных SQL на наличие записи в журнале по хешу этого рабочего элемента. Не найдя таковой, он выполняет вызов API и вставляет полученный номер счета-фактуры в журнал. После вставки создается запись Invoice, содержащая возвращенный номер счета-фактуры.

Если сервис нашел существующую запись в журнале, он вместо этого загрузит номер счета и создаст запись Invoice без вызова API. А если бы вставка не удалась из-за нарушения ограничений уникальности, она прервала бы обработку этого рабочего элемента, предупредив операторов о вероятном дублировании. На рис. 8.16 показана пара артефактов, поддерживающих шаблон «Исходящие».


**Рис. 8.16.** Сервисы совместно используют распределенный кеш и постоянную таблицу для поддержки шаблона Outbox

## Последствия

Не существует гарантированного механизма предотвращения дублирования вызовов API сторонних производителей. Даже при наличии этого шаблона дублирование иногда будет происходить. Нисходящие сервисы должны быть закодированы так, чтобы быть идемпотентными.

При запуске сервиса задержка в уведомлении других сервисов оставляет открытым окно, в котором оба сервиса могут считать себя первыми победителями. Для смягчения последствий параллельной обработки в этом сценарии отложите обработку новым сервисом до окончания обработки всех рабочих элементов другими сервисами.

Факт, сгенерированный из результатов вызова API, не должен содержать никакой информации, не зафиксированной в журнале. Если он содержит, например, информацию аудита, такую как метка времени или IP-адрес, когда и где он был записан, то факт будет отличаться от других фактов, представляющих тот же вызов API. Модель будет содержать повторяющиеся данные, даже если журнал предотвращает дублирование вызовов API.

## Связанные шаблоны

Для ручных процессов представьте пользователю очередь (*Queue*) с помощью простого запроса.

Рабочие элементы в папке «Исходящие» обычно являются транзакциями (Transactions).

Запрос к папке «Исходящие» чаще всего начинается с периода (Period). Перекрывайте периоды значительно больше, чем указано в SLA, чтобы предотвратить потерю рабочих элементов.

## Проектирование на основе ограничений

Представленные здесь шаблоны являются отправной точкой для создания приложений, использующих только неизменные исторические факты. Они имитируют настолько близко, насколько это возможно, поведение, которое люди привыкли ожидать от бизнес-приложений. И они делают это, используя только возможности неизменяемых распределенных данных.

Там, где эти шаблоны расходятся с ожидаемым поведением, они выявляют ограничения, связанные со средой, в которой они отображаются. Изменяемое свойство (*Mutable Property*) не может иметь единственное значение. И мы не можем требовать, чтобы сущность имела членство (*Membership*) только в одной группе. Эти слова говорят о том, что приложение распределено по нескольким узлам, каждый из которых имеет автономность, чтобы фиксировать параллельные изменения. Кроме того, мы не можем точно сказать, когда период (*Period*) закрыт. Мы можем только предположить, что прошло достаточно времени, чтобы удаленные узлы могли соединиться и поделиться своими рабочими элементами.

Мы не можем дать пользователям наших приложений именно то, чего они ожидают от централизованных систем. Правила неизменяемой архитектуры запрещают это. Причина проста – эти обещания *не могут* быть выполнены сильным, конечно согласованным образом. Архитектуры, которые тем не менее обеспечивают такое поведение, должны идти на компромисс с некоторыми аспектами своей распределенной природы, чтобы сделать это.

Приложение, построенное в соответствии с этими шаблонами, признает ограничения, накладываемые распределенными узлами. Оно начинает с этих ограничений и строится в направлении ожидаемого поведения, никогда не обещая больше того, что может быть разумно выполнено.

Эти шаблоны – не просто руководство по созданию распределенного приложения. Они являются средством коммуникации. Они делают возможным для разработчиков приложений говорить с заинтересованными сторонами об ограничениях без предварительного обучения их сильной конечной согласованности и о теореме CAP. Они позволяют нам говорить в общих чертах, не рассуждая о конкретных сценариях, в которых распределенные узлы могут вызвать у нас проблемы. Они задают рамки разговора о проектировании приложений, что помогает всем участникам установить ожидания и сохранить их.

# ЧАСТЬ III

# Реализация

# Глава 9

# Инверсии запросов

Шутка Матиаса Верраеса (Mathias Verraes) о двух трудных проблемах в распределенных системах основана на высказывании Фила Карлтона (Phil Karlton), бывшего сотрудника компании Netscape. Он сказал: «Есть только две трудные вещи в компьютерной науке: аннулирование кеша и именование вещей».

Я, как известно, не умею называть вещи (как вы, несомненно, обнаружили в предыдущих главах). Однако у меня есть надежное решение проблемы недействительности кеша. *Инверсии запросов* определяют не только то, какие кеши должны быть аннулированы, но и как именно они должны быть обновлены.

Когда мы думаем о кешировании, мы часто думаем о повышении производительности или масштабируемости. Но самым важным кешем является тот, на который смотрит пользователь. Пользовательские интерфейсы представляют собой кеш результатов запросов, временно хранящихся в моделях представлений и DOM браузера. Когда кеш становится недействительным, пользователь ожидает, что его представление автоматически обновится. Неспособность обновления пользовательского интерфейса приводит к разочарованию пользователя. Кнопка F5 в браузере и «потянуть, чтобы обновить» – это метафоры интерфейса, которые признают поражение.

На первый взгляд, кеширование неизменяемых данных должно быть простым. Если данные не меняются, кеш всегда актуален. Однако смысл кеша не в хранении копии неизменяемых фактов. Он предназначен для хранения копии результатов запросов. Именно результаты запроса, а не простые факты появляются в пользовательском интерфейсе. Поэтому мы должны найти стратегию для определения того, когда результаты запроса были затронуты.

Результаты запроса меняются по мере появления новых фактов. Для каждого нового факта мы должны ответить на два вопроса:

1. Какие кеши или компоненты пользовательского интерфейса были аннулированы?
2. Какие результаты должны быть добавлены или удалены из этих кешей и компонентов?

Инверсия запросов отвечает на один из этих вопросов, а иногда и на оба. Представления, которые затрагиваются, всегда могут быть найдены путем просмотра запроса от введенного факта назад. Результаты, которые нужно добавить или удалить, обычно можно определить по хвосту запроса от представленного факта вперед. И даже если инверсия не дает ответа на второй вопрос,

просто нужно повторно выполнить запрос, чтобы привести представление в актуальное состояние.

## Механизация проблемы

Определение того, что нужно обновить при наступлении определенных событий, – это хлеб и масло разработки приложений. Мы постоянно делаем это интуитивно. Реагируя на события пользовательского ввода, команды API или сообщения в очереди, разработчик решает, какое состояние является устаревшим. Это может быть модель представления, которая должна вызывать события изменения свойств, или массив результатов, который необходимо очистить и восстановить. Разработчики принимают такие решения.

Интуиция, однако, дает сбои в двух важных аспектах. Во-первых, легко пропустить зависимое состояние, которое необходимо обновить. Во-вторых, логику обновления следует пересматривать по мере добавления требований. Для решения первой проблемы мы просто тестируем до тех пор, пока нам не покажется, что мы нашли все причины изменения представления. А для второй проблемы мы анализируем каждую новую функцию на предмет того, как она взаимодействует с существующим поведением. Мы добавляем ее в качестве критерия приемки, изменяем все затронутые участки кода и тестируем на регрессии. Из-за этого изменения занимают больше времени и имеют больше шансов внести ошибки.

Лучшее решение – убрать решения об аннулировании кеша и обновлении пользовательского интерфейса из рук разработчика. Если бы система сама могла решать, какие кеши восстановить и какие компоненты пользовательского интерфейса обновить, то эта логика не загромождала бы код. Решения могут быть приняты механически, без риска человеческой ошибки. И они могут автоматически обновляться по мере роста системы, обеспечивая быстрое и безопасное добавление каждой новой функции.

Хорошей новостью является то, что система уже имеет достаточно информации, чтобы принимать решения о признании кеша недействительным. Каждый кеш и компонент пользовательского интерфейса первоначально заполняется с помощью запроса. Этот запрос описывает путь через данные системы, которые дают результаты для хранения или отображения. Обрабатывая запрос при его первом выполнении, система может определить, какие новые факты повлияют на результаты в будущем. Все, что нам нужно сделать, – это инвертировать запрос и начать отслеживать эти новые факты.

Давайте начнем с поиска формального словаря для описания запросов. Это поможет нам не только выполнять их, но и механически инвертировать.

## Анатомия запроса

Запрос, написанный на языке моделирования фактов (Factual Modeling Language), описывает типы и отношения между фактами. Он задает условия, при которых определенные факты должны быть в него включены. Запрос Factual – это декларативное заявление о результатах, которые мы ищем. Это описание должно быть переведено в нечто действенное.

Запрос, который мы изучали в главе 5, идентифицирует товары в корзине, за исключением тех, которые были заказаны:

```
query linesRemainingInCart(c: Cart) {
  match ol: OrderLine where ol.cart = c
       such that not exists o: Order where o.orderLines = ol
}
```

Мы наложили этот запрос на граф типов фактов, как показано на рис. 9.1.

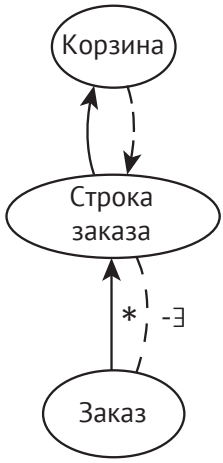


**Рис. 9.1.** Визуализация запроса в виде пути по графу фактов

Пунктирные стрелки указывают путь, по которому будет проходить граф экземпляров. Здесь описывается, как мы будем выполнять запрос. Мы можем формализовать этот процесс как конвейер. Он будет использоваться не только для выполнения запросов, но и для вычисления их инверсий.

## Последовательность шагов

Запрос – это последовательность шагов. Каждый шаг переходит от одного типа факта к другому по ребру. Ребро – это отношение предшественник/преемник между двумя фактами. Каждое ребро имеет роль – имя, данное предшественнику в определении типа преемника. Таким образом, мы можем определить шаг между фактами типа A и B как движение вверх к предшественнику или вниз к преемнику по ребру с заданной ролью.

Первый шаг в нашем примере запроса находит строки заказа, где `ol.cart = c`. Это подразумевает шаг-преемник. Учитывая корзину (cart), этот шаг находит все строки заказа-преемника с ролью `cart`. Мы можем нарисовать этот шаг, как показано на рис. 9.2.

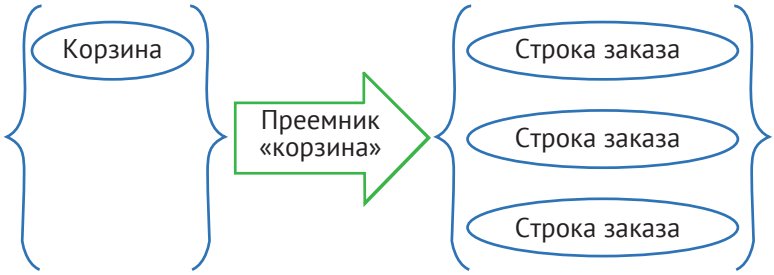


**Рис. 9.2.** На первом шаге конвейера набор корзин передается в набор строк заказа

Запрос – это путь от одного факта типа A к другому факту типа Z. Мы построим конвейер компонентов, каждый из которых принимает набор входных фактов и производит набор выходных фактов. Начальный набор содержит только один факт – начальную точку запроса.

## Фильтр по экзистенциальному состоянию

Некоторые компоненты конвейера вообще не будут шагами. Они не будут перемещаться вверх к предшественнику или вниз к преемнику по ребру. Вместо этого они будут фильтровать входной набор. Фильтры задаются в виде булева выражения экзистенциальных условий. Квантификатор каждого условия либо *существует*, либо *не существует*. Тело условия – это другой запрос, который начинается с типа факта, к которому применяется фильтр.

В примере запроса мы отфильтровываем все строки заказа, которые не являются частью заказа. Для этого мы используем экзистенциальное условие, основанное на подзапросе `where o.orderLines = ol`. Оно сохраняет только те строки заказа, для которых не существует заказа-преемника в роли `orderLines`. И вот мы добавляем компонент фильтрации, который загружает каждый факт в подчиненный конвейер. Он сохраняет только те факты, для которых подчиненный конвейер не дает результатов. Этот компонент показан на рис. 9.3.

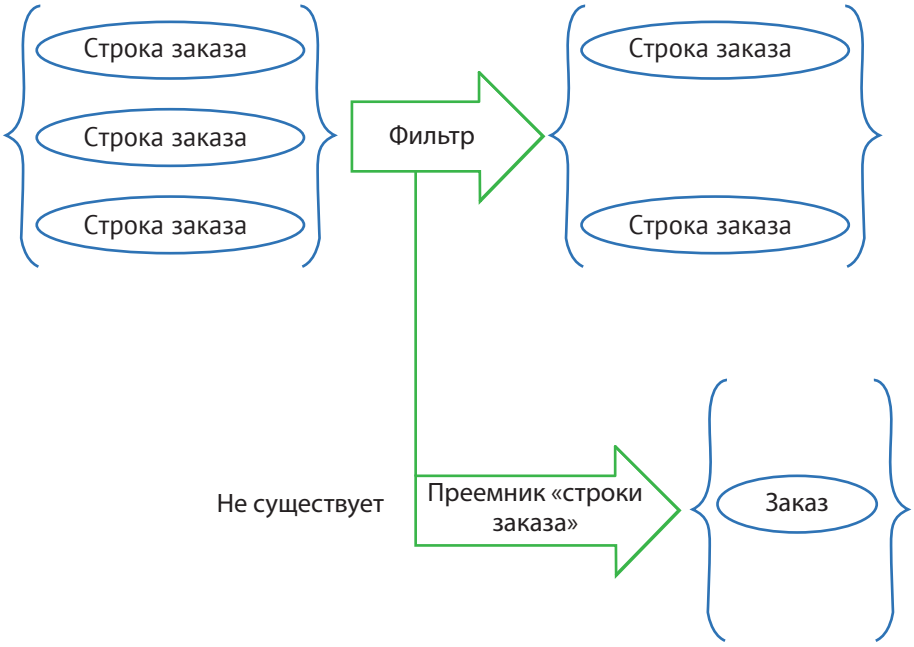


**Рис. 9.3.** На втором шаге происходит фильтрация строк заказа на основе подзапроса

Объединение этих двух половин в полный конвейер, как показано на рис. 9.4, дает нам механизм для вычисления запроса. Но это еще не все. Оно формализует шаги, которые мы до сих пор иллюстрировали пунктирными линиями, наложенными на диаграммы типов фактов.

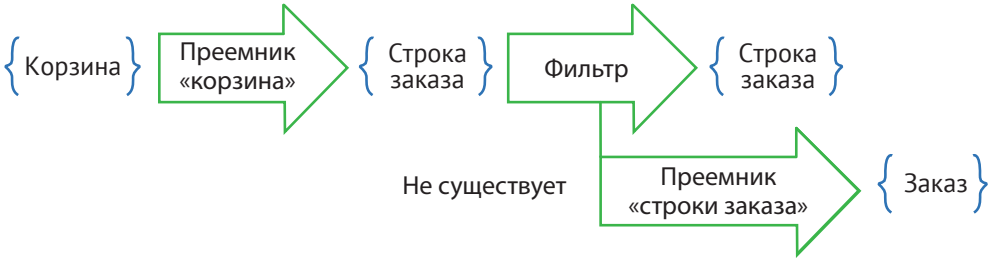


**Рис. 9.4.** Конвейер, состоящий из трех операций

Это компоненты, которые мы будем использовать для вычисления инверсии запроса. И начинается все с вычисления затрагиваемого набора. Чтобы понять, как это сделать, давайте вернемся к знакомым нам зигзагообразным путям.

## Затронутое множество

Первый вопрос, на который отвечает инверсия запроса, – на какие кеши и представления влияет новый факт. Он идентифицирует эти кеши и представления по начальной точке, из которой запрос был впервые выполнен.

Каждый запрос имеет начальную точку. Любой конкретный кеш или компонент пользовательского интерфейса хранит результаты запроса для определенного начального факта. Разные пользователи будут выполнять одинаковые запросы, но каждый из них будет начинаться с разного факта. Следовательно, их кеши будут разными. По мере навигации по пользовательскому интерфейсу кеши будут отображать компоненты, выполняя запросы. Хотя многие из этих компонентов будут выполнять один и тот же запрос, каждый из них будет начинаться с разных фактов. Начальная точка запроса определяет, какой компонент пользовательского интерфейса обновить или какой кеш считать недействительным.

В главе 5 мы показали пример представления, отображающего назначение столиков для официанта в ресторане. Для справки схема снова показана на рис. 9.5. Каждый официант войдет в систему и откроет одну и ту же страницу. Они будут видеть разные результаты, потому что компонент пользовательского интерфейса использует запрос, начинающийся с разных фактов. Когда вводится новый факт назначения (Assignment), система вычисляет затрагиваемого официанта (Server), чтобы определить, какое представление следует обновить.

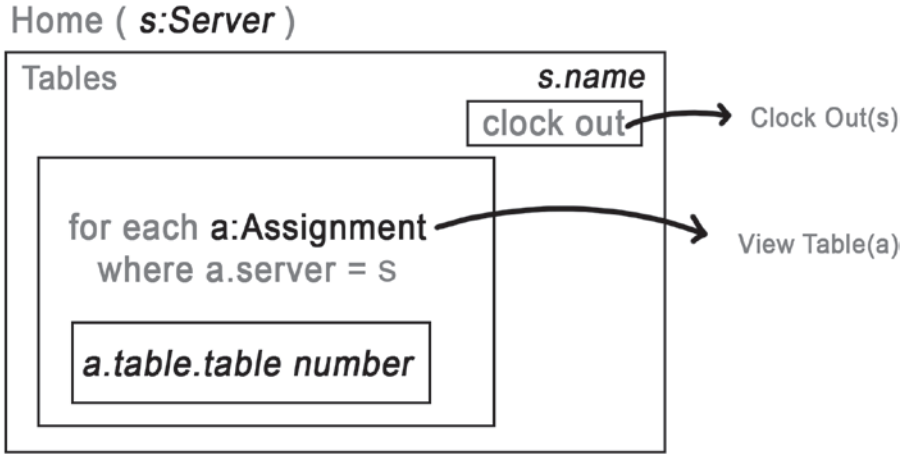


**Рис. 9.5.** Компонент пользовательского интерфейса, отображающий назначенные столики, начинается с факта официанта

## Вычисление затронутого набора

При первой загрузке представления списка столиков выполняется запрос, показанный на схеме. Этот запрос определяет, какие столики следует показать в списке. Но этот запрос также предоставляет всю информацию, необходимую для вычисления затрагиваемого набора при любом последующем назначении. Это и есть роль инверсии.

Оригинальный запрос, наложенный поверх графа ресторана, показан на рис. 9.6. Это путь от начального официанта (Server) к преемнику-назначению (Assignment).

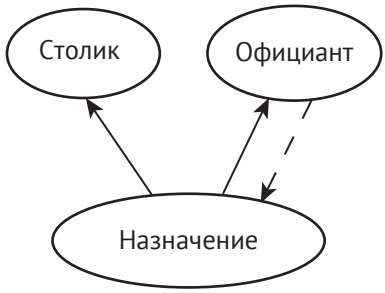


**Рис. 9.6.** Запрос назначения столика официанту – это один шаг к его преемнику

Инверсия – это просто обратный шаг, возврат от назначения к официанту. Когда вводится новое назначение, этот запрос сообщает нам, представление какого официанта нужно обновить. Это соответствует вашей интуиции и является точно такой же логикой, которую вы бы использовали, реагируя на событие назначения столика. Но, как показывает данный простой пример, процесс можно автоматизировать. Инверсный запрос показан на рис. 9.7.

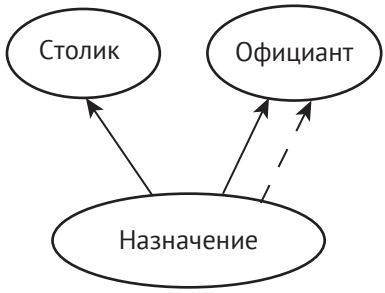


**Рис. 9.7.** Инверсный запрос просто выполняет тот же шаг в противоположном направлении

## Инвертирование длинных запросов

Предыдущий пример чрезвычайно прост. Он демонстрирует процесс инвертирования запроса, но не раскрывает всех нюансов. Когда мы начинаем изучать более длинные запросы, мы обнаруживаем, что у нас есть еще несколько соображений.

На запрос с одним шагом может повлиять только один вид введенного факта. Он имеет лишь одну инверсию. Но запрос с несколькими шагами может быть затронут, если введен любой из них. Более длинные запросы имеют больше инверсий. Таким образом, процесс инверсии запроса – это не просто инверсия цепочки шагов. Это создание всех обратных цепочек, которые указывают назад к начальной точке.

Рассмотрим более сложный сценарий. Мы создаем представление, которое отображает все компании, о которых должен позаботиться официант. В частности, это представление выбирает все назначения для официанта, затем для каждого столика выбирается компания, которая была рассажена. Этот многошаговый запрос показан на рис. 9.8.

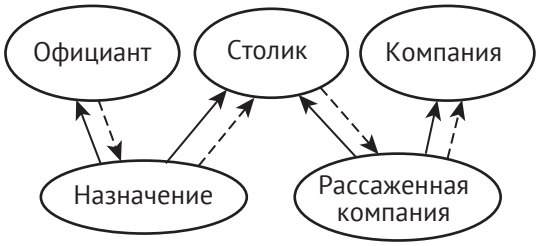


**Рис. 9.8.** Запрос из четырех шагов выдает все компании, сидящие за столиком, назначенным официанту

Есть несколько фактов, которые могут повлиять на это представление. Нам нужно получить инверсию для каждого шага на этом пути. Например, когда вводится новый факт «Рассаженная компания» (SeatParty), запрос на рис. 9.9 определяет, какие представления официанта будут затронуты.

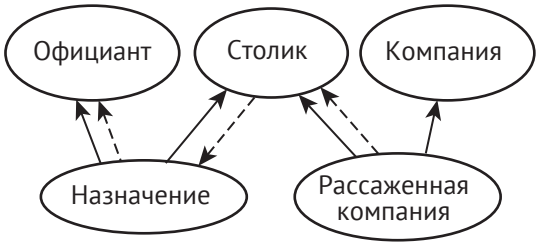


**Рис. 9.9.** Когда вводится факт «Рассаженная компания», затрагиваются все официанты, назначенные для данного столика

Каждая из инверсий представляет собой запрос от другого типа фактов обратно к официанту. Всего у этого запроса четыре инверсии. Они начинаются с каждого из типов «Назначение», «Столик», «Рассаженная компания» и «Компания» и становятся все длиннее, поскольку все заканчиваются на официанте.

## Неудовлетворительные инверсии

Инверсия запроса выполняется, когда вводится новый факт. Это может быть результат действия пользователя. Или новый факт может быть получен от какого-то другого источника. В любом случае, факт, который был введен, не присутствовал ранее и поэтому не представлен в результатах запроса. Это, в конце концов, и есть та самая причина, по которой мы обновляем представление.

Знание того, что новый факт только что прибыл в этот узел, говорит нам нечто весьма полезное. Это гарантирует нам, что у факта в настоящее время нет преемников.

Для того чтобы факт имел преемников, он должен быть представлен до того, как появятся преемники. Пользователь может выполнять только те действия, которые ссылаются на существующие факты. А чтобы одноранговое устройство могло поделиться фактом с этим узлом, оно должно сначала поделиться всеми своими предшественниками. Говоря более формально, факты вводятся в топологическом порядке.

Поскольку только что введенный факт еще не имеет преемников в системе, мы можем быть уверены, что любые запросы, начинающиеся с шага к преемнику, не дадут никаких результатов. В частности, любая инверсия, начинающаяся с шага вниз к преемнику, обязательно даст пустой затронутый набор. Такие инверсии являются *неудовлетворительными*. Они никогда не найдут представление, которое нужно обновить. Поэтому их можно игнорировать.

Пример неудовлетворительной инверсии показан на рис. 9.10. Когда вводится новый факт «Столик», мы можем быть уверены, что он еще не был назначен официанту. Поэтому мы можем предположить, что ни одно представление официанта не будет затронуто этим новым столиком. Мы будем игнорировать эту инверсию, так как она не будет иметь никакого эффекта.

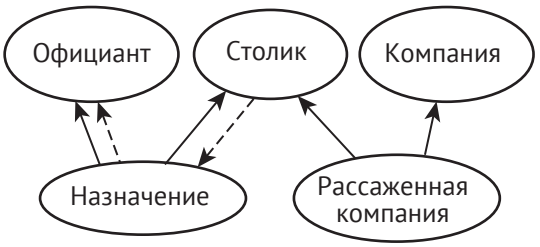


**Рис. 9.10.** Когда вводится новый столик, запрос для затронутых официантов начинается с шага к преемнику

Рассматривая эту оптимизацию визуально, рассмотрим исходный запрос как зигзагообразный путь по графу. У него есть холмы и долины, поскольку он поднимается к преемникам и опускается к предшественникам. Эта оптимизация говорит нам, что нужно рассматривать только те инверсии, которые начинаются в долинах. Если инверсия начинается на холме, то первым шагом инверсного запроса будет спуск к преемнику. Из четырех инверсий, которые мы вычислили из исходного запроса, только две являются удовлетворительными – те, которые начинаются с Назначения и Рассаженной компании.

## Движение назад

Теперь представим исходный запрос в виде конвейера. Конвейер начинается с набора Официантов (содержащих только начальный факт). Он заканчивается набором Компаний. По пути он делает четыре шага, попеременно к преемникам и предшественникам. Полный конвейер показан на рис. 9.11.


**Рис. 9.11.** Конвейер от набора официантов к набору компаний, которых они обслуживают

Чтобы вычислить инверсии этого запроса, возьмем каждый поднабор конвейера, начинающийся с официанта. Реверсируем каждый из шагов этого конвейера. Обратный шаг просто меняет преемника на предшественника,

и наоборот. В результате этой операции получаются четыре конвейера, показанных на рис. 9.12.

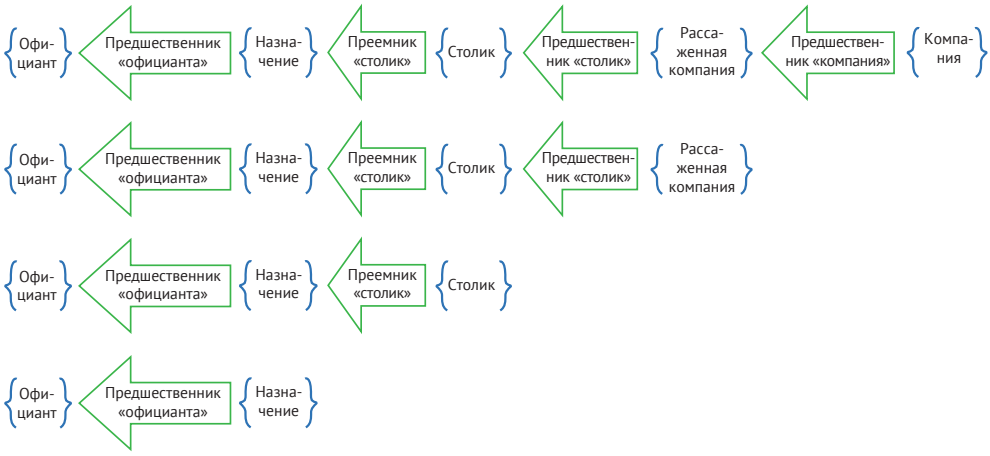


**Рис. 9.12.** Четыре инверсии исходного запроса имеют обратные шаги

Только две из этих инверсий являются удовлетворительными. Это те, которые начинаются с шага «Предшественник». Остальные две мы можем игнорировать. Полный набор удовлетворительных инверсий показан на рис. 9.13.

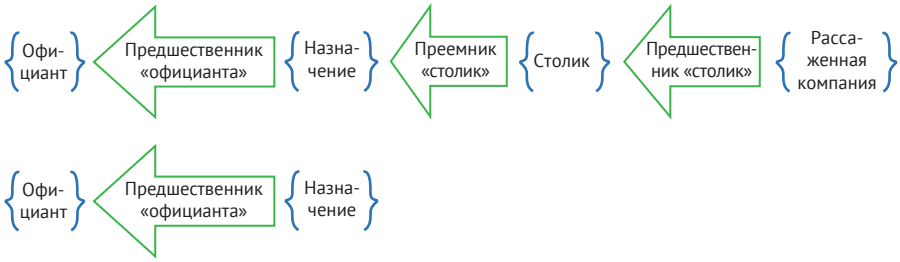


**Рис. 9.13.** Только две из инверсий являются удовлетворительными

Чтобы вывести список компаний, мы выполняем исходный конвейер, начиная с «Официанта». Затем каждый раз, когда вводится новый факт «Рассаженная компания», мы выполняем первый инверсный конвейер на рис. 9.13, чтобы найти затронутые представления. И всякий раз, когда вводится новый факт «Назначение», мы выполняем второй инверсный конвейер. Нет никаких других событий, которые могут повлиять на представление. Этот механический процесс находит все способы, которыми может быть затронуто представление.

## Доказательство полноты

Утверждение о том, что не существует других событий, которые могли бы повлиять на представление, нуждается в защите. Оно отнюдь не очевидно. Давайте уделим этому немного времени, прежде чем мы продолжим.

Для начала нам нужно определить, что мы подразумеваем под словом «событие». Событием может быть что-то, что сам пользователь сделал на этом узле, или информация, поступившая из другого узла. Если событие произошло на этом узле, то это может быть либо переходное действие, например перемещение, либо постоянное действие, например сохранение нового факта. Переходные действия приводят к новым отправным точкам для запросов и, следовательно, заменяют старые представления новыми. Постоянные действия всегда считаются введением новых фактов. Аналитический инструмент исторического моделирования не допускает никаких других постоянных действий, таких как изменения или удаления.

Поэтому каждое событие, которое может повлиять на существующее представление, будет являться введением факта. Это может быть факт, который пользователь ввел сам или который поступил от другого узла. В любом случае, если мы повторно выполним запрос к затронутому представлению, новый факт будет иметь определенное потенциальное влияние на результаты.

Наше утверждение о полноте эквивалентно утверждению о том, что инверсии находят все начальные точки, которые могут быть затронуты введенным фактом. Чтобы убедиться в том, что это так, мы исследуем наборы фактов, которые появляются на конвейере. Если факт появляется в одном из этих наборов, то он потенциально может повлиять на результаты конвейера. Если же нет, то он не может оказать влияния.

Обратите внимание, что реверсирование предшественника или преемника дает супернабор из начального набора. Если мы начинаем с одного набора фактов и находим всех преемников в данной роли, мы выбираем все ребра по роли и предшественнику. Двигаясь в обратном направлении, мы выбираем среди тех же ребер, но теперь по роли и преемнику. Ребра, которые дали результат на первоначальном шаге, будут среди тех, которые включены в обратный шаг. Таким образом, реверсирование предшественника или преемника даст все начальные факты, которые, вероятно, могут содержать данный конечный факт.

Пока что в этом доказательстве рассматриваются только шаги-предшественники и шаги-преемники. Мы завершим доказательство рассмотрением фильтров после того, как изучим экзистенциальные условия.

## Новые результаты

Знание того, какое представление нужно обновить, – это отличное начало. Как только мы узнаем, что представление затронуто, мы можем просто запустить его запрос снова с начальной точки, чтобы найти новое состояние. Но часто можно сделать лучше. Иногда инверсия запроса подсказывает нам, какие именно новые результаты будут добавлены в представление.

Мы проанализировали запрос для компаний, которых ожидает официант. Две инверсии, которые мы обнаружили, инвертируют только часть исходного запроса. Начиная с введенного факта, эти части приводят нас обратно к затронутому набору. Оставшаяся часть запроса, однако, ведет нас к новым резуль-

татам. Нам не нужно выполнять потенциально затратный запрос. Поскольку мы уже знаем введенный факт, нам нужно выполнить только часть запроса, чтобы найти новые результаты. Остаток каждого инвертированного запроса показан на рис. 9.14.

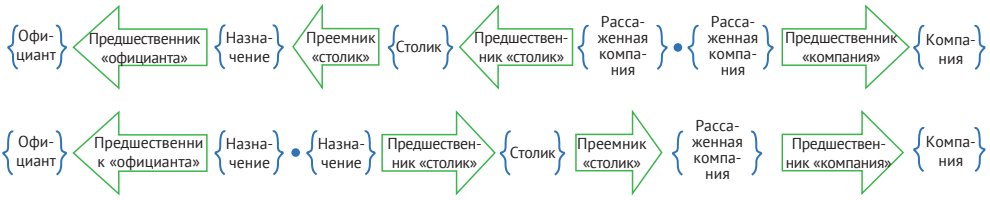


**Рис. 9.14.** Оставшаяся часть запроса (показанная после точки) дает новые результаты для добавления к затронутому представлению

Чтобы выполнить эти инверсии, начните с введенного факта. Выполните обратный конвейер, чтобы найти затронутый набор. Затем выполните прямой конвейер, чтобы найти новые результаты.

Например, когда компания усаживается, вводится новый факт «Рассаженная компания». Запросите всех официантов, назначенных на столик, за который усадили компанию. К каждому из их представлений добавьте компанию. И когда вводится новое назначение столика, следуйте по конвейеру в обратном направлении до соответствующего официанта. Обновите представление этого официанта со всеми компаниями, сидящими в настоящее время за этим столиком.

Вероятно, ваша интуиция заставила вас написать именно такой код. Возможно, вы поняли, что не нужно заново выполнять весь запрос, чтобы обновить представление. Но, следуя механическому процессу для автоматического создания оптимального алгоритма, вы получите уверенность в том, что ни один крайний случай не был упущен и никакие новые функции не повредят этот код.

## Дальнейшая оптимизация

Как и раньше, мы знаем, что введенный факт не имеет преемников. Точно так же, как мы использовали это наблюдение для удаления неудовлетворительных инверсий, мы можем определить неудовлетворительные результаты. Если прямой запрос начинается с присоединения преемника, то вся инверсия может быть исключена. Возможно, мы сможем найти некоторые затронутые представления, но мы знаем, что этот новый факт не приведет к новым результатам.

Возьмем абстрактный пример на рис. 9.15. Мы хотим вывести представление, начинающееся с факта A, которое отображает информацию о фактах E. Запрос проходит зигзагами по графу, перескакивая от предшественника к преемнику и от преемника к предшественнику.

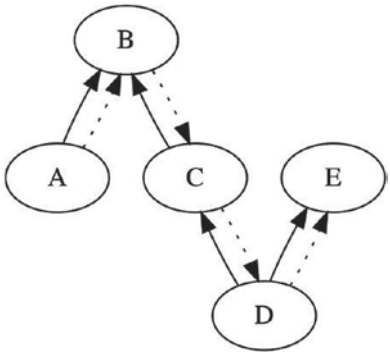


**Рис. 9.15.** В некоторых запросах может потребоваться несколько шагов преемника подряд

Вычисляя инверсию в каждой точке пути, мы обнаруживаем, что обратный конвейер из C начинается с шага-предшественника. Сначала может показаться, что он удовлетворителен. Однако обратите внимание, что прямой конвейер из C в D начинается с шага-преемника. Следовательно, эта инверсия неудовлетворительна. Только инверсия, начинающаяся в D, дает новые результаты для затронутого представления. Инверсии всегда начинаются с долин графа типов фактов.

## Экзистенциальные условия

До сих пор мы рассматривали простые запросы. Они представляют собой последовательность шагов, каждый из которых направлен к предшественнику или преемнику. Однако большинство запросов, используемых для наполнения пользовательского интерфейса или для наполнения кеша, являются более сложными. Они состоят не только из шагов-предшественников и шагов-преемников, но и условий `exists` и `not exists`. Давайте добавим эти операторы к нашим базовым инверсиям и посмотрим, как они будут работать.

Более простым из двух видов экзистенциальных условий является `exists`. Мы пока оставим оператор `not exists` в стороне. Этот вид условия фильтрует факты только до тех, для которых последующий запрос дает результат. Вспомним пример из главы 8.

```
query pickedOrders(c: Company) {
  match o: OrderSubmission where o.company = c
      such that exists ps: PackingSlip where ps.orderSubmission = o
}
```

Чтобы начать процесс, мы преобразуем этот запрос Factual в конвейер. Предложение `where o.company = c` подразумевает шаг-преемник. Начиная с `Company` он ищет преемников `OrderSubmissions` в роли «`company`». Предложение `such that` начинает подзапрос. Подзапрос проверяет существование `PackingSlip`, используя другой шаг-преемник – `where ps.orderSubmission = o`. Соединяя эти операции вместе, мы получаем конвейер, показанный на рис. 9.16.

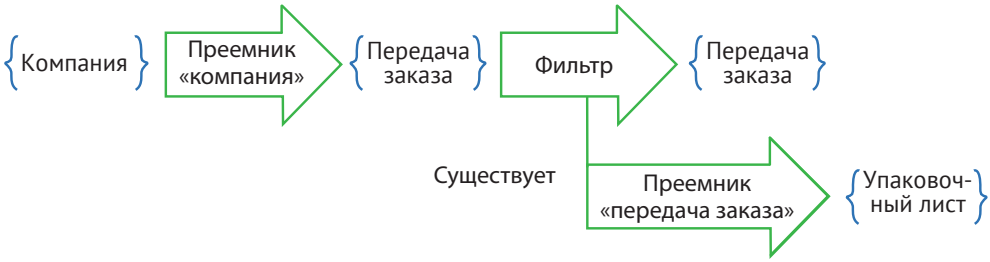


**Рис. 9.16.** Конвейер, имеющий условие exists

Получив этот конвейер, мы можем выполнить запрос. Набор передач заказов, выходящий из фильтра, будет поднабором входящих. Каждая из `OrderSubmission` передается на последующий шаг, и только те из них, которые дают результат, сохраняются. Рассмотрим, что происходит с результатами, когда вводится новый `PackingSlip`. Если он увеличивает количество результатов подзапроса с 0 до 1, то это приводит к тому, что его родитель `OrderSubmission` появится в результатах.

## Рекурсивная инверсия

Чтобы разбить этот конвейер так, чтобы его можно было инвертировать механически, лучше всего применить рекурсию. Как только мы достигнем условия `exists`, мы можем переместить верхнюю часть конвейера в стек. Это сфокусирует наше внимание только на нижней части конвейера – подзапросе, показанном на рис. 9.17.

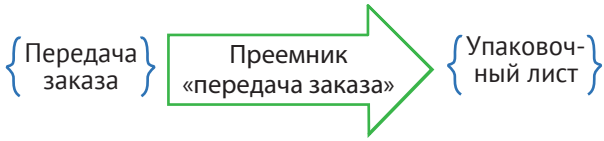


**Рис. 9.17.** Точкой входа подзапроса является тип, над которым работает фильтр

Мы знаем, как инвертировать этот запрос: `PackingSlip` влияет на своего предшественника `OrderSubmission`, как показано на рис. 9.18. Новым результатом является сам упаковочный лист. Поскольку затронутый конвейер не начинается с шага-преемника (он начинается с предшественника в роли «`orderSubmission`»), мы знаем, что затронутый набор является удовлетворительным. А поскольку оставшийся конвейер не начинается с шага-преемника (это просто нулевой конвейер), мы знаем, что новые результаты также удовлетворительны. Таким образом, `PackingSlip` влияет на `OrderSubmission`, вводя новый результат.

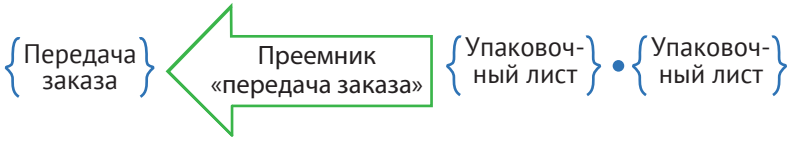


**Рис. 9.18.** Инверсия подзапроса добавляет введенный упаковочный лист к затронутой передаче заказа

Открыв стек и вернувшись к верхней части, мы приходим к выводу, что введение `PackingSlip` приводит к введению `OrderSubmission`. Мы приходим к такому выводу на основании квантификатора `exists`. Передача заказа без последующего упаковочного листа будет отфильтрована. Поэтому введение упаковочного листа добавляет его в конвейер.

Мы можем открыть стек и продолжить анализ конвейера, как если бы это был `OrderSubmission`, а не `PackingSlip`. То есть мы можем, за исключением одной небольшой оговорки. Поскольку `OrderSubmission` уже был известен этому узлу, у него могут быть и другие преемники. Поэтому мы не можем применить оптимизацию долины, которую использовали ранее. Мы должны сохранить инверсию, даже если условие существования не появляется в долине графа. Оптимизация долины применяется только к внутреннему подзапросу.

Инверсия этого запроса представлена на рис. 9.19. Мы начинаем с возвращения от представленного упаковочного листа к затронутой им передаче заказа. Члены этого набора теперь будут удовлетворять условию родительского запроса. Отсюда затронутый набор всего запроса является обратным конвейером, исходя из этого условия. А новые результаты – это оставшийся прямой конвейер.

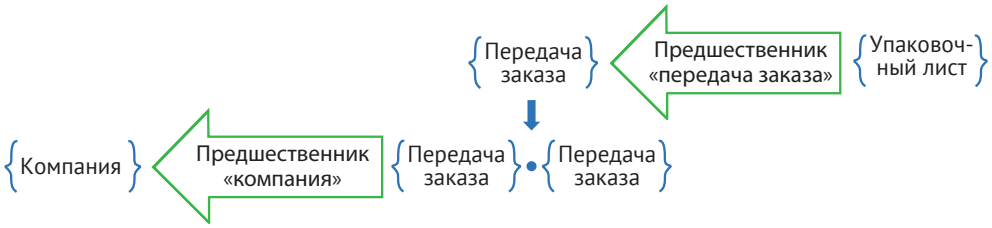


**Рис. 9.19.** Инверсия начинается с инверсии подзапроса и продолжается так, как если бы был введен затронутый факт

## Условия хвоста

В предыдущем примере нас не волнует упаковочный лист, который был добавлен к результатам подзапроса. Нам важен только сам факт добавления результатов. Инверсия подзапроса, показанная на рис. 9.18, показывает добавленный конвейер после точки. Но мы отбросили эту деталь как несущественную на рис. 9.19.

В целом, однако, эта деталь имеет значение. Подзапрос говорит нам, какое условие фильтра может быть затронуто введенным фактом. Однако он не гарантирует нам, что условие действительно истинно. Чтобы иметь такую гарантию, мы должны знать, что введенный факт привел к некоторым фактическим дополнениям. Вот почему мы, как правило, хотим выполнить хвост подзапроса. Условие фильтра, на которое оказывается воздействие, гарантированно будет истинным только в том случае, если хвост дает некоторые результаты.

Давайте добавим еще несколько шагов к исходному запросу, чтобы проиллюстрировать происходящее. Предположим, что мы хотим знать не только наличие упаковочного листа, но и адрес доставки получателя. Пересмотренный конвейер запросов теперь выглядит так, как показано на рис. 9.20.

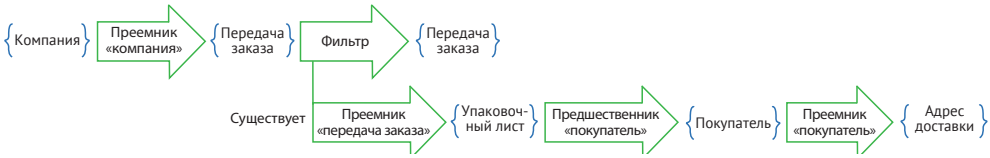


**Рис. 9.20.** Пересмотренный конвейер имеет два дополнительных шага в подзапросе

Как и раньше, мы обрабатываем этот конвейер рекурсивно. Подзапрос имеет две инверсии – одну, начинающуюся на упаковочном листе, и одну, начинающуюся на адресе доставки. Мы будем рассматривать только первую. Этот обратный подзапрос имеет нетривиальный хвостовой конвейер, как показано на рис. 9.21.

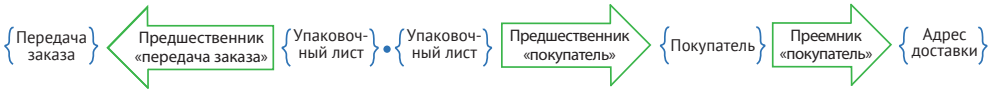


**Рис. 9.21.** Инверсия подзапроса сохраняет два новых шага в хвосте

Когда мы сталкиваемся с этой инверсией, мы знаем, что она добавляет результаты в подзапрос только тогда, когда хвостовой конвейер выдает результат. Только тогда он сделает условие `exists` истинным. Чтобы отфильтровать передачи заказа, для которых это условие стало истинным, мы превратим хвостовой конвейер в фильтр, как показано на рис. 9.22.

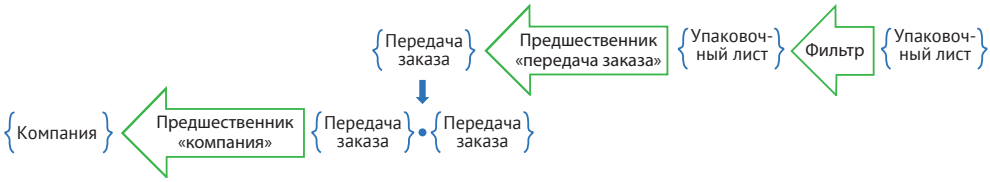


**Рис. 9.22.** Хвост подзапроса становится условием фильтра

В общем, мы всегда должны включать хвост инверсии подзапроса `exists` в качестве условия фильтрации. В некоторых случаях это условие фильтрации оказывается тавтологией. Оно всегда будет удовлетворяться, как в примере с тривиальным хвостовым конвейером. Когда условие является тавтологией, фильтр можно не оптимизировать.

## Удаление результатов

Предложение `exists` является более простым из двух экзистенциальных квантификаторов. Оно имеет только эффект введения новых результатов. Другое предложение, `not exists`, допускает новую возможность. Это предложение может удалять результаты. Последствия гораздо более значительны, чем просто отрицание условия.

Предложения `not exists` встречаются на практике чаще, чем предложения `exists`. Причина этого проста – только условие `not exists` может удалять резуль-

таты из представлений. Без них представления могли бы расти беспредельно и быстро становиться непригодными для использования. Давайте вернемся в главу 3 к одному из первых запросов, которые мы рассмотрели.

```
query gamesInProgress(p: Player) {
  match g: Game where g.player = p
       such that not exists w: Win where w.game = g
}
```

Конвейер для этого запроса имеет уже знакомую структуру, показанную на рис. 9.23. В последний раз, когда мы видели эту структуру, фильтр использовал квантификатор `exists`. Теперь он использует квантификатор `not exists`. Выполнение этого конвейера происходит так же, как и раньше, за исключением того, что условие отрицается. Фильтр сохраняет только те факты партии, по которым подзапрос не дает результатов.

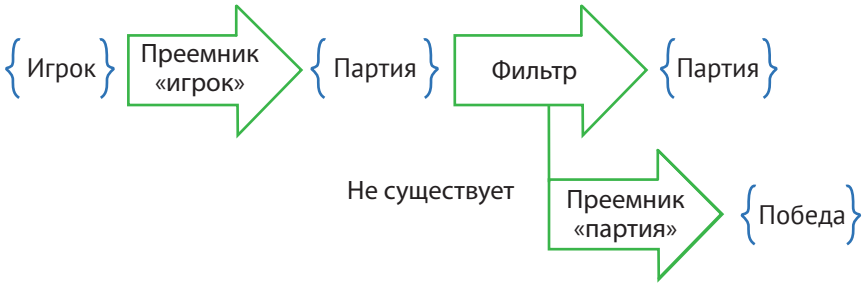


**Рис. 9.23.** Предложение not exists фильтрует на основе подзапроса

Самое большое различие – в инверсии. Введение победы (Win) приводит к тому, что партия (Game) удаляется из отфильтрованного набора. Это распространяется на внешний запрос и подразумевает, что партия удаляется из общих результатов. Как показано на рис. 9.24, мы должны отслеживать эффект каждого подзапроса. Мы должны указать, добавляет он или удаляет затронутый факт.

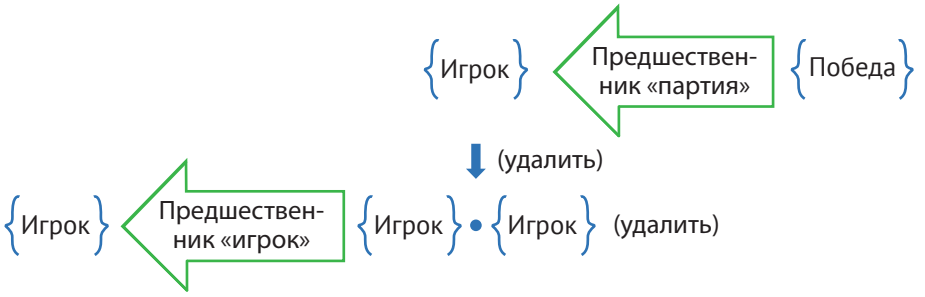


**Рис. 9.24.** Инверсия предложения not exists приводит к удалению результатов во внешнем конвейере

## Когда удаление не является удалением

Инверсия, которая удаляет результаты, существенно отличается от инверсии, которая их добавляет. Это не просто вопрос смены знака. Различие заключается в том, что мы можем гарантировать. Мы не всегда можем знать, что результат, который инверсия хочет удалить, действительно будет удален. Доказать отрицание гораздо сложнее.

Когда результат добавляется в затронутый запрос, мы можем гарантировать, что новый результат будет действительно возвращен, если мы выполним запрос снова. Мы хотели бы сделать аналогичное утверждение об удаленных результатах. Мы хотели бы утверждать, что удаленные результаты *не* будут возвращены из запроса. К сожалению, это не всегда возможно.

Предположим, что запрос ищет не текущие партии, а их противников. Запрос Factual в этом случае будет выглядеть следующим образом:

```
query opponentsInGamesInProgress(p: Player) {
  match g: Game where g.player = p
        such that not exists w: Win where w.game = g
  then o: Player where g.player = o
}
```

Конвейер для этого запроса выглядит так же, как и для предыдущего, с одним дополнительным шагом. После фильтрации партий он делает шаг-предшественник в сторону игрока, как показано на рис. 9.25. В данном конкретном случае это будет включать и стартового игрока, но это не тот вопрос, который нас волнует. Мы можем легко отфильтровать стартового игрока, чтобы получить осмысленное представление.

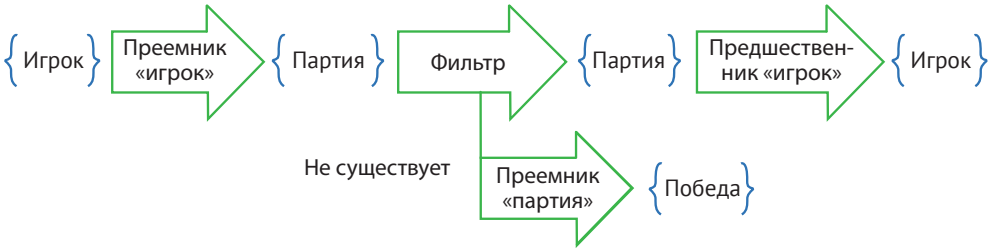


**Рис. 9.25.** Конвейер для поиска текущих противников делает дополнительный шаг-предшественник к игроку

Проблема, которая нас беспокоит, заключается в том, что инверсия имеет нетривиальный конвейер «удаления». Он делает шаг, которого не делает инверсия предыдущего запроса. Мы видим этот дополнительный шаг предшественника на рис. 9.26.

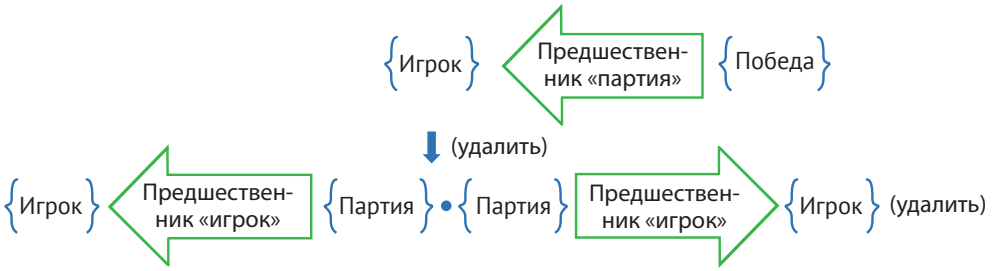


**Рис. 9.26.** Инверсия удаляет игрока удаленной партии

Этот дополнительный шаг заставит нас удалить противника из только что закончившейся партии. Когда вводится победа, партия спускается вниз и удаляет противника из затронутого представления. Однако возможно, что протекает другая партия с этим же противником. Существует более одного способа пройти по графу, чтобы добраться до этого противника. Удаление одного из этих путей не гарантирует, что других путей не существует.

Предыдущий запрос был безопасным. Мы могли гарантировать, что любая партия будет удалена из кеша, когда вводилась победа. Независимо от того, как мы проходили граф, фильтр применялся к самой партии. Но если граф выходит за пределы фильтра, то мы больше не можем дать такую гарантию. Если набор удалений представляет собой нетривиальный конвейер, то мы должны проверить все возможные пути.

В подобных ситуациях удаление не может гарантировать, что удаленный результат все еще не является действительным результатом по какому-то другому пути. Как следствие мы должны выполнить исходный запрос снова. Затронутый набор говорит нам, какие кеши следует аннулировать. Но набор remove не говорит нам, какие результаты нужно удалить. В лучшем случае он дает набор кандидатов, которых мы можем проверить.

## Вложенные подзапросы

В процессе обработки вложенных подзапросов нам нужно будет отслеживать, добавляют ли они или удаляют результаты. Нередко подзапрос `not exists` может иметь свой собственный подзапрос `not exists`. Когда это происходит, введение факта во внутренний подзапрос *добавит* результат в представление. Каждый рекурсивный спуск в подзапрос `not exists` меняет направление эффекта.

Типичным примером вложенного подзапроса с использованием условия `not exists` является шаблон *Восстановление*, рассмотренный в главе 8. Согласно этому шаблону, введение факта `EntityDeletion` удаляет сущность из результатов запроса, а последующее введение факта `EntityRestore` восстанавливает ее.

```
query entitiesInOwner(o: Owner) {
  match e: Entity where e.owner = o
     such that not exists ed: EntityDeletion where ed.entity = e
       such that not exists er: EntityRestore where er.deletion = ed
}
```

Запрос становится конвейером с вложенными шагами фильтрации, как показано на рис. 9.27. Оба этих шага применяют квантификатор Not Exists.

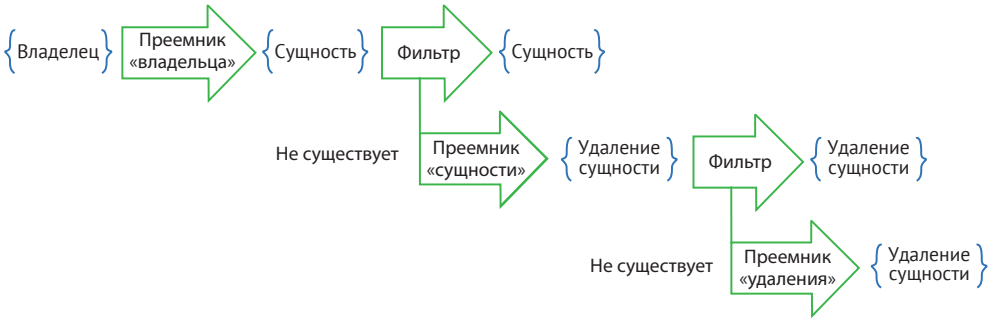


**Рис. 9.27.** Шаблон восстановления дает конвейер с вложенными фильтрами not exists

Чтобы найти инверсии этого запроса, мы обходим конвейер рекурсивно. Спустившись на два уровня, мы инвертируем самый внутренний запрос. Это говорит нам о том, что введение `EntityRestore` влияет на предшествующее `EntityDeletion`. Оно имеет эффект удаления этой сущности. Это эффект, который мы можем гарантировать, поскольку мы инвертировали условие `not exists` и можем доказать, что преемник существует.

Открыв стек, мы определяем эффект от удаления `EntityDeletion`. Инверсия говорит нам о затронутом наборе фактов сущности. Удаление факта удаления сущности в идеале должно иметь эффект *добавления* сущности. Однако такой эффект мы не можем гарантировать.

Возможно, было еще одно удаление сущности. Конечно, мы можем доказать, что удалили одну из них. Но мы еще не доказали, что удалили их все. Поэтому, прежде чем идентифицируем добавленные сущности, мы повторно выполняем исходное условие фильтрации. Полная инверсия показана на рис. 9.28.

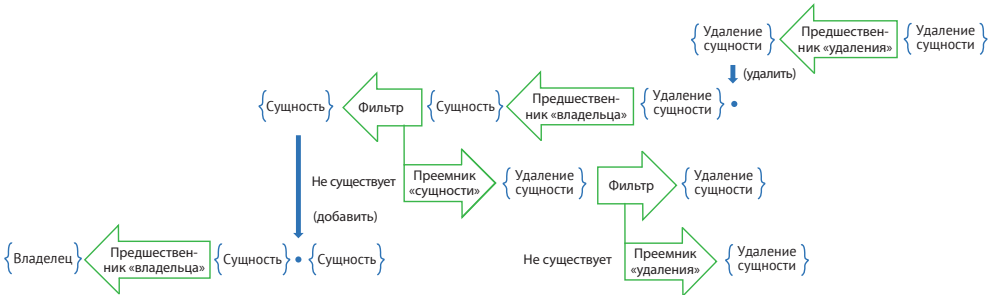


**Рис. 9.28.** Инверсией двух вложенных предложений not exists является конвейер, который вызывает добавление фактов

В общем случае мы можем гарантировать эффект инверсии на основе добавления факта. Внутренний подзапрос всегда имеет такую форму, поскольку он фиксирует введение нового факта. Но любая инверсия, основанная на удалении факта, не может быть гарантирована. Могут существовать другие пути

вдоль графа, которые сохраняют условие в прежнем виде. Поэтому мы всегда должны добавлять к инверсии фильтр, проверяющий исходное условие.

## Тавтологические условия

В предыдущих примерах с инверсиями я упустил важную деталь. Инверсия должна сохранять экзистенциальное условие исходного запроса. Я опустил ее как для ясности, так и потому, что выбрал примеры, в которых это не имеет значения. В каждом из предыдущих примеров экзистенциальное условие гарантированно истинно при тех обстоятельствах, в которых оно было применено. Логический предикат, который всегда истинен, называется *тавтологией*. Когда мы распознаем тавтологии в инверсиях, мы можем избавиться от них.

Первым примером экзистенциального условия, который мы рассмотрели, был запрос на собранные заказы. Этот пример был неполным. Полный запрос из главы 8 включал дополнительное предложение, ищущее запрос на доставку (RequestForDelivery). Это условие исключало заказы с отгрузкой.

```
query pickedOrdersToShip(c: Company) {
  match o: OrderSubmission where o.company = c
     such that exists ps: PackingSlip where ps.orderSubmission = o
     and not exists rd: RequestForDelivery where rd.orderSubmission = o
}
```

В то время как человек-аналитик понимает мотивацию запроса, механический инвертор запросов не понимает. Вместо того чтобы полагаться на нашу интуицию относительно заказов с доставкой, давайте инвертируем этот запрос механически. На этот раз мы сохраним условия инверсии и будем искать тавтологии. Мы начнем с построения конвейера из исходного запроса, как показано на рис. 9.29.

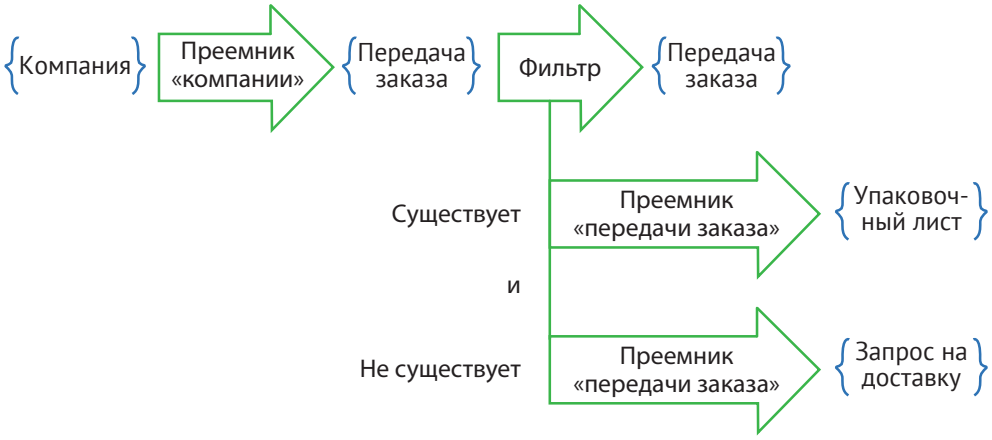


**Рис. 9.29.** Запрос с составным условием определяет конвейер с двумя подзапросами

Этот конвейер имеет несколько инверсий, которые механический инвертор произвел бы путем рекурсивного прохождения каждого из шагов в каждом

подзапросе. Как и раньше, мы сосредоточимся только на одной инверсии – той, которая описывает реакцию на введение PackingSlip. Однако на этот раз мы сохраним фильтр внутри инверсии. Полный конвейер инверсии показан на рис. 9.30.

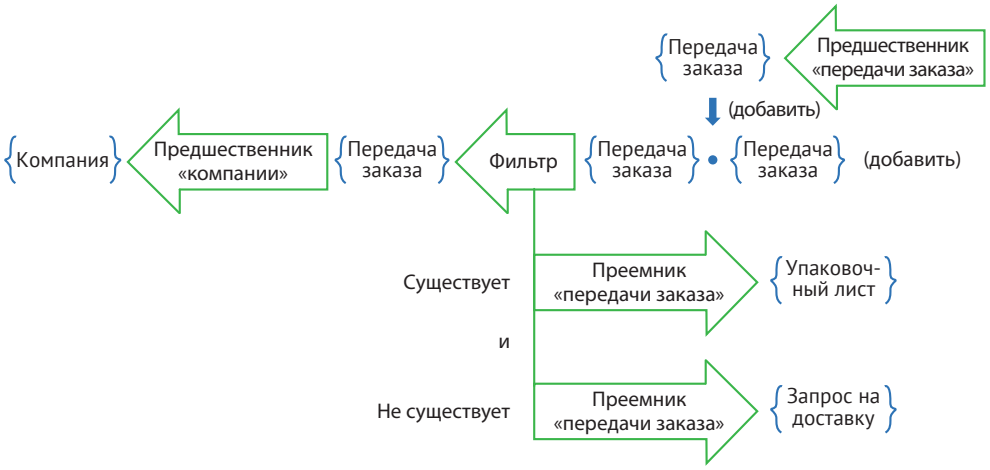


**Рис. 9.30.** Инверсный конвейер сохраняет экзистенциальные условия исходного запроса

На этом этапе мы можем оптимизировать конвейер. Мы знаем, что конвейер начинается с введения упаковочного листа. Передачи заказа, которые спускаются во вторую линию, имеют преемника упаковочного листа. Поэтому мы знаем, что первый экзистенциальный запрос должен быть удовлетворен. Определенно существует преемник упаковочного листа в роли `orderSubmission`. Это именно тот упаковочный лист, который инициировал процесс. Поэтому мы можем исключить это условие из фильтра, как показано на рис. 9.31.

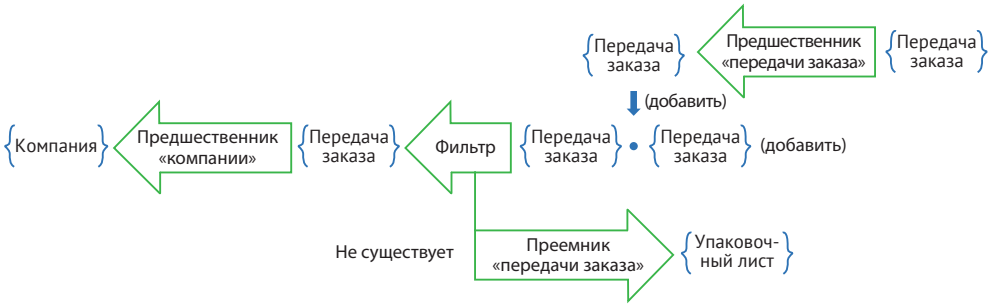


**Рис. 9.31.** Мы можем исключить тавтологические условия из фильтра

Конвейер не дает нам никаких знаний об оставшемся условии. Поэтому мы должны проверить его, прежде чем добавить результирующую передачу заказа. В первый раз, когда мы рассматривали эту инверсию, условия `RequestForDelivery` не было в конвейере. Поэтому в итоге мы получили фильтр, содержащий только тавтологическое условие. Мы оптимизировали фильтр, так как он не имел никакого эффекта.

Полученная инверсия подразумевает определенную процедуру. Когда появляется упаковочный лист, добавьте заказ в кешированные результаты, но только в том случае, если он не был отправлен напрямую. Ведь аналитик мог бы спросить: «Зачем нам создавать упаковочный лист для заказа, отгруженного напрямую с фабрики?», поскольку это было бы абсурдно. Это замечание может заставить разработчика исключить подобную проверку. Но у механического инвертора нет таких предубеждений. Он будет производить код, который логически корректен, даже если он не имеет смысла для бизнеса. И как результат он будет работать правильно в нелепых крайних случаях, когда решение, основанное на интуиции, не сработает.

В общем, мы найдем два сценария, при которых условие является тавтологией. Во-первых, у нас будут существовать предложения, ищущие факты, которые только что были введены. И во-вторых, будут предложения `not exists`, ищущие преемников только что введенных фактов. Только что рассмотренный пример является иллюстрацией первого сценария. Второй является расширением оптимизации долины. Мы исключили инверсии, которые начинались за пределами долины. Эти инверсии начинались с шагов-преемников либо в затронутом наборе, либо в новых результатах. Таким же образом мы можем считать, что любое предложение `not exists` в начале инверсии, начинающейся с шага-преемника, является тавтологией.

Когда мы отменяем `not` в двух общих случаях, упомянутых ранее, мы находим другую оптимизацию. Противоположностью тавтологического предиката является *неудовлетворительный* предикат. Возможно, что инверсия будет содержать предложение `not exists`, ищущее введенные факты. Учитывая, что инверсия срабатывает только тогда, когда эти факты введены, мы знаем, что эти условия являются ложными. Также возможно, что инверсия будет содержать предложение `exists` для преемников введенных фактов. Опять же, поскольку эти факты были введены недавно, эти условия должны быть ложными. Неудовлетворительные условия внутри оператора *or* (или) могут быть устранены, а находящиеся внутри оператора *and* (и) делают неудовлетворительным весь оператор. Механический инвертор запросов будет оптимизировать и эти операторы, и даже оптимизирует всю инверсию, если условие фильтра будет признано неудовлетворительным.

## Продолжение доказательства полноты

Теперь, когда мы лучше понимаем влияние фильтров на инверсию запроса, мы можем закончить доказательство полноты. Мы уже показали, что инверсия, состоящая только из шагов-преемников и шагов-предшественников, не исключает ни одной возможной затронутой начальной точки. Теперь мы покажем то же самое для инверсий, содержащих фильтры.

Шаг-предшественник или шаг-преемник обладает тем свойством, что инверсия производит супернабор исходного начального набора. Мы полагались на это, чтобы показать, что все начальные точки, которые включают введенный факт, будут находиться в этом супернаборе. Компоненты фильтра не обладают этим свойством. Фильтр в любом направлении дает поднабор исходного набора. Для фильтров мы должны учитывать, добавляем мы факт или удаляем его.

Если мы добавляем факт, например для самого внутреннего подзапроса, отвечающего на введенный факт, тогда мы можем утверждать, что добавленный факт будет иметь эффект на запрос, если он прошел через фильтр в исходном конвейере. Если он был отфильтрован, то он не будет иметь никакого последующего эффекта. Поэтому применение фильтра в обратном направлении, даже если оно производит поднабор, производит именно тот поднабор, который действительно будет затронут.

С другой стороны, если мы удаляем факт, то эта логика не работает. Мы ищем затронутый набор начальных точек, чьи конвейеры раньше содержали удаленный факт, но больше не содержат. Поэтому фильтр будет исключать именно те начальные точки, которые затронуты удаленным фактом. Отмена фильтра не решает проблему. Затронутый набор может быть отфильтрован более чем по одной причине, и любой из оставшихся фильтров может ошибочно исключить эти точки.

Тогда решение состоит в том, чтобы удалить все фильтры из затронутого набора удаленных инверсий. Единственный фильтр, который должны иметь эти инверсии, – это фильтр в конце, который проверяет, что условие фильтра действительно ложно для этих затронутых начальных точек. Любой другой фильтр уменьшит затронутый набор и потенциально пропустит эффекты.

## Потенциальные и фактические изменения

Рассуждая о влиянии экзистенциальных условий, мы сделали предположение. Мы предположили, что эффектом инверсии подзапроса всегда будет изменение условия фильтрации с *false* (ложь) на *true* (истина) и, таким образом, добавление результата, или с *true* (истина) на *false* (ложь) и, таким образом, его удаление. Теперь, когда мы осознали всю сложность инверсий подзапросов, тавтологий и неудовлетворительных предикатов, мы можем признать, что это предположение не является истинным. Как только мы это сделаем, мы сможем увидеть, какой простой дополнительный шаг нам нужно сделать для защиты от тех случаев, когда оно ложно.

Впервые мы представили это предположение при рассмотрении предложения `exists`. Там мы сказали, что если введенный факт увеличивает количество результатов подзапроса с 0 до 1, то условие становится истинным, и новые результаты добавляются. Затем мы перешли к поиску инверсий подзапроса, которые, как мы можем с уверенностью утверждать, увеличат количество результатов подзапроса. Однако у нас не было доказательств того, что они увеличивают количество результатов с 0 до 1. Если подзапрос уже имел результаты, то новый подзапрос не имел бы заметного эффекта. Условие фильтрации уже выполнено.

Продолжая действовать так, как будто новый факт увеличил количество результатов подзапроса с 0 до 1, мы сделали предположение, что новые результаты, которые он добавляет, не были уже результатами исходного запроса. Поскольку это предположение неверно, мы можем обнаружить, что добавляем результаты в кеш или представление, которые уже присутствуют. Таким образом, следствием принятия этого предположения для выражения `exists` является дублирование.

## Удаление отсутствующих результатов

Аналогичное предположение возникло, когда мы рассматривали предложение `not exists`. Здесь мы перешли к оценке инверсий условия в предположении, что любой новый факт приведет к увеличению числа результатов с 0 до 1. Если рассматривать предложение `not exists`, то это изменит условие фильтрации с *true* на *false* и, следовательно, удалит результаты. Но если в подзапросе уже были результаты, то еще один факт не сделает условие фильтрации более ложным. Эффектом выполнения инверсии в этих обстоятельствах будет попытка удалить результат, которого не было в кеше.

У этой проблемы есть простое и сложное решение. Позвольте мне сначала описать сложный вариант, он интересен, но в конечном итоге является бессмысленным. Поскольку граф фактов является неизменяемым, введение нового факта не уничтожает информацию. У нас все еще есть возможность видеть граф таким, каким он был до введения факта. Мы можем вычислить условие фильтрации в том виде, в котором оно было до введения факта, который вызвал инверсию.

Если мы запустим условие фильтра – предложение `exists` или `not exists`, игнорируя при этом новый факт, мы можем увидеть, был ли он раньше ложным. Если да, то мы знаем, что введение нового факта как раз и сделало его истинным. Мы будем знать, что набор подзапросов изменился с 0 на 1, а значит, эффект реален.

Это решение интересно тем, что является конкретным примером мощной техники анализа. Оно демонстрирует, что мы можем ответить на гипотетические вопросы «что, если», выполняя запросы и игнорируя поднаборы направленного ациклического графа. Эти вопросы могут дать представление о том, как система выглядит на разных узлах. Они дают нам мощную возможность отладки «путешествия во времени», популярную в таких фреймворках, как Elm и Redux. Но, увы, для решения проблем дублирования кеша и удаления отсутствующих результатов есть гораздо более простое решение.

## Кеши есть множества

Результаты запроса Factual всегда являются множествами. Они не навязывают порядок и не допускают дубликатов. Поэтому любой кеш или представление, созданное запросом, будет иметь в своей основе множество. Он может далее применять порядок поверх этого множества или проецировать множество через некоторую агрегатную функцию, но то, с чего он начинал, был неупорядоченный набор отдельных фактов.

До тех пор, пока кеш или представление сохраняет набор фактов, он может легко защитить от предположения, что каждое потенциальное изменение является фактическим изменением. Если инверсия сообщает, что введенный факт добавляет множество результатов, кеш просто выполняет объединение множеств. При этом игнорируются любые результаты, которые уже есть в множестве. Он добавляет только те, которые действительно являются новыми. Аналогично, для обработки инверсии, которая удаляет результаты, кеш выполняет разность множеств. Он удаляет только те результаты, которые были в множестве изначально.

При создании представления очень заманчиво просто отслеживать проецируемый DOM, XAML или другие визуальные элементы. И если пользователей кеша интересует только изменяемая проекция некоторых фактов, а не сами необработанные неизменяемые результаты, то вполне логично, что изменяемая проекция – это все, что нужно хранить в кеше. Однако простейшим решением проблемы аннулирования кеша является хранение множества, которое приводит к проекции. В ответ на инверсию запроса выполните объединение или разность множеств. А затем спроецируйте ту часть, которая изменилась.

## Инверсия запросов на практике

Как вы только что убедились, инверсия запросов – дело непростое. Существует несколько крайних случаев, сценариев и оптимизаций, которые необходимо учитывать. К счастью, этот процесс можно автоматизировать. Разработчикам не нужно вручную вычислять инверсию запроса.

Механический процесс инверсии запроса гарантированно дает затронутый набор, который определяет все возможные недействительные кеши. Более того, он часто производит добавление или удаление набора, которое точно описывает, какое объединение или разность множеств нужно вычислить. Но даже когда инверсия слишком сложна для получения таких точных инструкций, повторное выполнение запроса даст правильные результаты.

В прошлом я создал несколько реализаций алгоритма инверсии запросов. Первая была для проекта с открытым исходным кодом под названием Correspondence. Вторая – для более позднего проекта с открытым исходным кодом под названием Jinaga. Оба проекта выводили инверсии из исходных запросов разработчика, чтобы определить, какие представления следует обновлять при введении новых фактов. Их поведение было удивительным для меня, даже при знании стоящей за ними математики. И я обнаружил, что моя уверенность и производительность значительно повысились благодаря наличию инвертора запросов, работающего от моего имени.

После публикации этой книги я буду поддерживать справочник с открытым исходным кодом по реализации алгоритма инверсии запросов. С помощью читателей я смогу найти крайние случаи и оптимизации, которые упустил в этой главе. Эти улучшения станут частью эталонной реализации. Любые разработчики проектов, желающие использовать инверсию запросов, смогут учиться на этой реализации и сравнивать результаты при помощи формы автоматизированного тестирования. Вы можете найти ссылку на эту эталонную реализацию на сайте Apress по адресу immutablearchitecture.com.

# Глава 10

# Базы данных SQL

Неизменяемые модели – это все об ограничениях. Приложение не должно изменять данные. Ему запрещено удалять или перезаписывать информацию. Эти ограничения являются аксиомами, на которых строится математическая структура неизменяемой архитектуры. Обеспечение соблюдения этих ограничений является обязанностью каждого уровня стека приложений, вплоть до системы хранения данных.

Неизменяемая модель может быть поддержана любым типом хранилища данных. Одной из самых мощных, гибких и популярных форм механизма хранения является реляционная база данных. Перевод ограничений неизменяемой модели в реляционную базу данных требует дисциплины. Мы должны избегать некоторых возможностей, которые предоставляют реляционные базы данных. Для начала мы должны запретить использование `UPDATE` и `DELETE`. Эти две операции являются частью языка манипулирования данными (*Data Manipulation Language* – DML). Только неразрушающая DML команда `INSERT` может использоваться в неизменяемом хранилище данных.

Ограничение DML – это хорошее начало, но мы должны сделать еще больше, чтобы хранить неизменяемую модель в реляционной базе данных. Мы определили запросы к неизменяемой модели в соответствии с определенными ограничениями. Они должны начинаться с известного факта. Они могут проходить по графу только по отношениям предшественник/преемник. И они допускают лишь экзистенциальные условия, данные не могут быть отфильтрованы по значению. Эти ограничения ограничивают наше использование языка запросов *Data Query Language* (DQL). Язык DQL включает в себя мощные и многофункциональные ключевые слова `SELECT`, `WHERE` и `JOIN`. При хранении неизменяемой модели в реляционной базе данных, однако, мы должны ограничиться подмножеством шаблонов.

Чтобы лучше работать с этим подмножеством DQL, мы также ограничим использование языка определения данных (*Data Definition Language* – DDL). Операторы DDL создают таблицы (`TABLE`), индексы (`INDEX`)и другие объекты. Мы обнаружим подмножество DDL, которое лучше всего подходит для определения схемы для неизменяемых записей.

Со всеми этими ограничениями может показаться, что мы теряем мощь и гибкость реляционных баз данных. Но на самом деле мы получаем гораздо больше взамен. Подмножество, которым мы ограничимся, позволит

создать чрезвычайно эффективный уровень доступа к данным. А там, где эффективность может быть повышена, мы найдем повторяющиеся модели оптимизации. Шаги, которые мы выполняем для создания DDL, DML и DQL, будут чрезвычайно механическими. Они будут настолько предсказуемы, что мы даже сможем автоматизировать некоторые или все из них по генерации кода SQL. Если дойти до крайности, то мы даже найдем способ перемещать модель данных из одного хранилища в другое, развертывать новый код без изменения схемы базы данных и даже решить проблему нелинейного версионирования данных.

Для начала мы научимся писать DDL для определения реляционной схемы, хранящей неизменяемую модель. Подмножество, которое мы будем использовать, тщательно выбрано, чтобы соответствовать ограничениям неизменяемой архитектуры. Проектирование в соответствии с этими ограничениями может показаться на первый взгляд лишением возможностей. Но, как мы увидим, это действительно дает нам возможности, которых мы никогда не могли бы иметь раньше.

# Идентичность

Когда мы впервые представили исторические записи в качестве данных приложения, то узнали, что запись однозначно идентифицируется по значениям ее полей и идентичности ее предшественников. Хотя это определение идентичности полезно для доказательства теорем об идемпотентности и сходимости историй, оно не очень полезно для проектирования баз данных. Если бы каждая строка идентифицировалась по сумме ее столбцов, то внешний ключ был бы полной копией родительской таблицы. Это было бы совершенно непрактично.

Реляционная база данных требует, чтобы мы идентифицировали исторические записи с помощью некоторой формы суррогатного ключа. Наиболее естественным выбором для большинства систем реляционных баз данных являются автоинкрементные идентификаторы. Они обеспечивают наиболее оптимальную эффективность хранения и производительность запросов. Однако у них есть недостаток, заключающийся в том, что они привязаны к конкретному местоположению, что подробно обсуждалось в главе 4. Поэтому мы опишем механизм использования автоинкрементных идентификаторов для их внутренних преимуществ, но сопоставим их с идентификаторами на основе содержимого, с целью преодоления недостатков.

## Хранение с адресацией по содержанию

Концептуально содержание исторической записи – это и есть ее идентичность. Однако на практике нам нужен идентификатор для содержимого. Решение состоит в том, чтобы получить идентификатор из содержимого с помощью хеш-функции. Эта практика известна как *хранение с адресацией по содержанию*. Данный подход работает только для неизменного содержимого. Поэтому он является подходящим механизмом для идентификации неизменяемых записей.

На первый взгляд, это кажется нелепостью. Чтобы загрузить запись, вы должны знать ее адрес. Но чтобы найти ее адрес, нужно сначала узнать ее содержимое. Кажется, что вы никогда не сможете начать работу. На практике, однако, найти отправную точку совсем не сложно. Если кто-то делает запись, то он знает ее содержание. Но если кто-то читает запись, то он получает ее хеш. Отправная точка просто меняется в зависимости от того, читаете вы или пишете.

Рассмотрим блокчейн, в котором транзакция должна ссылаться на цифровой документ. Место на самом блокчейне стоит слишком дорого, что запрещает нам хранить документ внутри транзакции. Вместо этого мы храним хеш. Фактический документ будет храниться в гораздо более дешевой распределенной базе данных, такой как InterPlanetary File System (IPFS)[1].

Автор документа обладает оригинальным содержимым и поэтому может вычислить хеш. Поскольку документ неизменяем, этот хеш никогда не изменится и поэтому может быть разумным суррогатом идентичности документа. Читатель транзакции, с другой стороны, не имеет оригинального документа. Он не может вычислить хеш, чтобы определить идентичность. Однако это не страшно, потому что он может прочитать идентификатор из транзакции.

Поиск документа в распределенной файловой системе сводится к поиску объекта по его хешу. Получив его, читатель может проверить идентификатор документа, вычислив его хеш. В той мере, в какой он может быть уверен, что будет трудно создать поддельный документ с таким же хешем, он может быть уверен, что этот неизменяемый документ не был подделан.

## Преимущества

Помимо устойчивости ко взлому при использовании хеша документа в качестве идентификатора, у нас есть еще несколько преимуществ. Некоторые из них уже были рассмотрены в главе 4. Другие будут новыми. Теперь, когда у нас есть понимание того, как моделировать систему с использованием неизменяемых записей, давайте потратим немного времени, чтобы оценить эти преимущества.

Как уже отмечалось в примере с документом в распределенной файловой системе, пишущий вычисляет хеш перед сохранением записи. Это подразумевает, что еще до общения с другим узлом автор записи знает ее идентификатор. Узлу нет необходимости связываться с каким-то внешним сервисом, прежде чем он узнает идентичность. Это дает ему возможность продолжать работу, даже создавать другие связанные записи и ссылаться на эту идентификацию, прежде чем ему понадобится подключиться к одноранговому узлу.

Есть также преимущество в том, что каждый узел будет вычислять одну и ту же идентификацию для одной и той же записи. Это обеспечивает естественное преимущество дедупликации. Представьте, если бы ваши фотографии идентифицировались по хешу их содержимого, а не по папке, которую они занимают, и имени файла, которое им присвоила камера. Если бы это было так, то вы никогда не смогли бы случайно создать дубликаты фотографий, повторно импортируя их с цифровых носителей или обмениваясь ими с друзьями. Такое

[1] https://ipfs.io.

же преимущество существует и в бизнес-приложениях. Предотвращение дубликатов обеспечивает идемпотентность и позволяет избежать дублирования эффекта транзакции.

Наконец, рассмотрим проблему объединения данных из разрозненных узлов. Если идентификатор записей был основан на чем-либо, кроме их содержимого, то две вещи могут пойти не так. Во-первых, возможно, что две записи с одинаковым содержанием будут иметь разные идентификаторы, каждый из которых присвоен другим узлом. Во-вторых, две разные записи могут случайно иметь общий идентификатор. Обе эти проблемы затрудняют определение того, где структуры данных пересекаются, а где расходятся.

При использовании хранения с адресацией по содержанию каждый узел вычисляет идентификатор совершенно одинаково. Если две записи имеют общий идентификатор, то это одна и та же запись. А если это одна и та же запись, то они имеют один и тот же идентификатор. Если используется любой другой механизм, то идентификаторы должны быть сопоставлены по мере перемещения данных из одного пространства идентификаторов в другое.

## Коллизии хешей

Когда хеш используется в качестве замены содержимого записи, необходимо решить одну важную проблему – коллизии хешей. Идентификатор записи уникален. Ее идентификатор должен отличаться от идентификаторов других записей. Если содержимое двух разных записей имеет один и тот же идентификатор, то ссылки на одну из них можно спутать со ссылками на другую. Учитывая принцип «голубятни», мы точно знаем, что коллизии возможны для любой записи, размер которой превышает размер хеша. Тогда вопрос заключается в том, возникнет ли когда-нибудь на практике любая из этих других записей, дающих тот же хеш.

Одна из возможностей заключается в том, что недобросовестный субъект *намеренно* создаст такую запись, что позволит ему подделывать подписи и подменять документы. Для борьбы с этим мы полагаемся только на криптографически сильные хеши, которые разработаны и постоянно тестируются для предотвращения такой деятельности. Криптографы создавали все более сложные алгоритмы хеширования с постоянно увеличивающимся размером хеша в гонке вооружений против этих атак.

Другая возможность заключается в том, что коллизия произойдет *по чистой случайности*. В наборе данных с $n$ записей существует $n(n-1)/2$ их пар. Мы должны позволить хешу иметь достаточно возможных значений, чтобы ограничить вероятность того, что *любая из этих пар* имеет одинаковый хеш. Если немного упростить, то число пар почти равно $n^2/2$, поэтому число возможных хешей $h$ должно быть значительно большим, чем это число. Вероятность коллизии очень точно аппроксимируется следующим уравнением:

$$p = 1 - e^{-n^{2/2h}}.$$

Используя это приближение, мы можем определить риск коллизии. Пусть количество битов в $n$ равно $s$, а количество битов в $h$ равно $b$. Это позволяет нам

вычислить количество битов в вероятности коллизии. Вероятность коллизии $p = 1 / 2q$, где $q$ определяется следующим образом:

$$q = b - 2s + 1.$$

Вывод этой формулы см. на боковой панели. Например, если у нас есть набор данных с миллиардом записей и мы используем хеш SHA-1, то можем вычислить вероятность возникновения коллизии *в любом месте набора данных* следующим образом:

$$1 \text{ миллиард записей} \approx 2^{30};$$

$$\text{размер хеша} = 160 \text{ бит};$$

$$\text{вероятность коллизии} \approx 1 \text{ из } 2^{160-2\cdot30+1} = 1 \text{ из } 2^{101}.$$

Таким образом, при использовании хеша SHA-1 вероятность столкновения в любом месте набора данных объемом 1 млрд записей составляет примерно 1 к $10^{30}$. Если мы улучшим хеш до SHA-2 с 256 битами, вероятность упадет до 1 к $10^{59}$. Для любого реалистичного набора данных этот риск исчезающе мал.

## ВЕРОЯТНОСТЬ КОЛЛИЗИИ ХЕШЕЙ

Начиная с аппроксимации вероятности коллизии хеша, мы можем упростить задачу:

$$p = 1 - e^{-n^{2/2h}},$$
$$1 - p = e^{-n^{2/2h}},$$
$$ln(1 - p) = -n / 2h.$$

Учитывая, что мы рассматриваем только сценарии, в которых вероятность близка к нулю, мы можем применить приближение малого значения, что $ln(1 + x) = x$:

$$p = n^2/2h.$$

Заменяя величины их определениями в степени двойки, получаем:

$$2^{-q} = (2^s)^2/2\cdot2^b$$
$$2^{-q} = 2^{2s}/2^{b+1}$$
$$2^{-q} = 2^{2s-b-1}$$
$$-q = 2s - b - 1$$
$$q = b - 2s + 1$$

Эта формула говорит нам о том, что при каждом удвоении набора данных нам необходимо увеличивать размер хеша на 2 бита, чтобы сохранить низкую вероятность коллизий.

### Избегайте использования хешей в качестве первичных ключей

Хеши – это фантастические идентификаторы. Однако они являются плохими первичными ключами. Самая очевидная причина – размер. Большинство современных компьютерных систем имеют естественный размер слова в 64 бита. Реляционные системы управления базами данных используют преимущества этого размера слова для своих собственных типов и структур данных. Хеши, с другой стороны, обычно имеют размер 160, 256 или 512 бит, в зависимости от выбранного алгоритма. Более крупные идентификаторы могут быть заметно медленнее и более громоздкими в управлении.

Большие первичные ключи могут привести к раздуванию журналов, так как записи журнала должны использовать весь ключ для ссылки на строки. Это влияет не только на хранение, но и на производительность. Выполнение транзакции включает запись потока журнала. Репликация требует отправки его на другие машины. Обе эти операции требуют времени.

Большие идентификаторы также приводят к раздуванию индексов. Сканирование индекса с большими идентификаторами означает загрузку большего количества страниц для поиска нужных данных. Перестройка индексов занимает больше времени. Кроме того, операции запроса не могут выполняться на границе слов машины. В результате общая производительность запросов снижается.

Наконец, большинство реляционных систем баз данных используют первичный ключ в качестве кластеризованного индекса. Это означает, что физическое хранение записей будет упорядочено в соответствии с этим ключом. Наиболее оптимальным способом выделения места для следующей записи является добавление ее в блок. Наименее оптимальный способ – вставить ее случайным образом где-то в середине. Хеши выглядят случайными. Использование хеша в качестве кластеризованного индекса приведет к фрагментации и частому разбиению страниц.

## Структура таблицы

Мы можем применить эти знания для ограничения языка определения данных Data Definition Language с целью создания хорошо управляемых неизменяемых моделей. Чтобы получить преимущества хранения с адресацией содержимого и сохранить эффективность автоинкрементных идентификаторов, мы будем хранить и то, и другое. Автоинкрементный идентификатор будет первичным ключом таблицы. Его значение, однако, никогда не покинет базу данных. Для внешнего мира будет казаться, что каждая запись идентифицируется только по ее хешу. Мы наложим ограничение уникальности на хеш, чтобы обеспечить необходимую защиту от дублирования.

Поскольку хеш больше, чем размер слова в компьютере, большинство движков баз данных не отображают его в собственные типы данных. Вместо этого они обычно хранятся в столбцах в двоичном или текстовом формате. Если хеш хранится в виде текста, он кодируется в формате base-64[1]. Это облегчает его

[1] Base64 — стандарт кодирования двоичных данных при помощи только 64 символов ASCII. Алфавит кодирования содержит текстово-цифровые латинские символы A–Z, a–z и 0–9 (62 знака) и 2 дополнительных символа, зависящих от системы реализации. Каждые 3 исходных байта кодируются 4 символами (увеличение на $^1/_3$). – *Прим. перев.*

просмотр оператору, но увеличивает требования к хранению на ⅓ часть. В следующих примерах мы будем использовать текстовый вариант, поскольку он более удобен. Но, пожалуйста, подумайте, будет ли двоичный вариант лучше для вашего приложения.

Давайте вспомним один из самых первых задокументированных исторических фактов. Это был факт Catalog, не имеющий ничего, кроме естественного ключа, чтобы отличить его от других каталогов. Преобразуем его в таблицу SQL.

```
fact Catalog {
  referenceNumber: string
}
```

Таблица для факта будет содержать первичный ключ, хеш и столбец для каждого неизменяемого поля. В данном примере есть только одно поле, поэтому результирующая таблица содержит всего три столбца. Вот как эта таблица была бы определена в PostgreSQL.

```
CREATE TABLE catalog (
  catalog_id SERIAL PRIMARY KEY,
  catalog_hash VARCHAR(88) NOT NULL UNIQUE,
  reference_number VARCHAR(50) NOT NULL
);
```

Причина выбора 88 символов заключается в том, что в данном приложении используется 512-битный SHA-2 хеш. В кодировке base-64 512-битное число занимает 88 символов ASCII. Чтобы найти это значение для любого размера хеша, разделите количество битов на 24, округлите, а затем умножьте на 4.

Заманчиво определить ограничение уникальности, охватывающее поля таблицы. Мы знаем, что значения этих полей идентифицируют факт, и поэтому никакой другой факт не может иметь эти значения. В предыдущем примере мы, возможно, захотим определить ограничение уникальности для номера ссылки. Хотя это сработает для данного конкретного примера, на практике это не работает. Мы увидим это более наглядно при рассмотрении предшественников с кардинальностью «*много*». Во всех случаях ограничение уникальности хеша служит той же цели. Поэтому мы будем полагаться только на это ограничение.

Чтобы вычислить хеш записи, найдем каноническую форму. Я предпочитаю использовать JSON с отсортированными по алфавиту полями и удаленными пробельными символами. Вычислим хеш в формате UTF-8 кодированного строкового представления.

```
$ echo -n '{«reference_number»:»AX247»}' | \
> openssl dgst -sha512 -binary | \
> base64
8rC0hVD...ifdw==
```

Такая структура таблицы позволяет выполнять вставку без необходимости ждать, пока первичный ключ будет сгенерирован. Вызывающая сторона уже знает хеш записи, и это все, что требуется для уникальной идентификации строки. Когда позднее потребуется catalog_id, его можно будет получить по catalog_hash.

Если приложение попытается вставить дубликат записи, ограничение уникальности предотвратит это. Нет причин рассматривать это как ошибку. Приложение просто подтвердит, что запись была сохранена. Большинство движков реляционных баз данных позволяют игнорировать нарушение ограничения уникальности. В PostgreSQL, например, используйте предложение «ПРИ КОНФЛИКТЕ НИЧЕГО НЕ ДЕЛАТЬ» (ON CONFLICT DO NOTHING).

```
INSERT INTO catalog
  (catalog_hash, reference_number)
  VALUES ('8rC0hVD...ifdw==', 'AX247')
  ON CONFLICT DO NOTHING;
```

## Отношения

Когда мы переводим отношения предшественников в реляционные таблицы, они становятся внешними ключами. Чтобы получить преимущества пространства и производительности целых чисел, внешний ключ будет ссылаться на автоинкрементный ID предшественника, а не на хеш. Это позволяет сохранить наши индексы малыми и быстрыми. Но внешние ключи не покидают базу данных.

Отношения предшественников имеют три степени кардинальности: один, необязательно (0 или 1) и много (0 или более). Если отношение допускает ровно одного предшественника, то мы представляем его непосредственно как внешний ключ в таблице фактов. Например, мы можем перевести продукт, находящийся в каталоге, в таблицу. Определение продукта в Factual выглядит следующим образом:

```
fact Product {
  catalog: Catalog
  sku: string
}
```

Полученная таблица имеет автоинкрементный ID, хеш, поля и внешние ключи для предшественников. Обязательно создайте индекс для внешнего ключа, так как большинство реляционных систем баз данных не делают этого автоматически.

```
CREATE TABLE product (
  product_id SERIAL PRIMARY KEY,
  product_hash VARCHAR(88) NOT NULL UNIQUE,
  catalog_id INT NOT NULL REFERENCES catalog,
  sku VARCHAR(50) NOT NULL
);
```

```
CREATE INDEX product_catalog
  ON product (catalog_id);
```

## Вставка преемников

Теперь, когда схема определена, мы можем определить, как лучше структурировать наш язык модификации данных. При вставке строк-преемников вызывающая сторона не знает идентификатор (ID) предшественника. Это происходит потому, что ID никогда не покидал базу данных. Это было важным проектным решением для защиты независимости от местоположения. Как следствие нам необходимо, чтобы база данных искала ID предшественника всякий раз, когда она выполняет вставку.

Мы будем искать ID предшественника, используя хеш предшественника. Выполнение поиска в операторе вставки `INSERT` требует использования синтаксиса `INSERT ... SELECT`. Вызывающее приложение вычисляет хеши предшественника и факта преемника. Запрос ищет идентификатор предшественника по хешу и вставляет хеш преемника напрямую.

```
INSERT INTO product
  (product_hash, catalog_id, sku)
  SELECT 'fK02Oge...5GFw==', catalog_id, 'PK47'
     FROM catalog
     WHERE catalog_hash = '8rC0hVD...ifdw=='
  ON CONFLICT DO NOTHING;
```

Чтобы вычислить хеш преемника из канонической формы, необходимо использовать хеш предшественника. В каноническом JSON, который я предпочитаю, предшественник представлен объектом с полем `ref`. Значением этого поля является хеш предшественника. Предшественники сортируются в алфавитном порядке среди остальных полей.

```
$ echo -n '{«catalog»:{«ref»: «8rC0hVD...ifdw==«}, «sku»: «PK47»}' | \
> openssl dgst -sha512 -binary | \
> base64
fK02Oge...5GFw==
```

## Необязательные предшественники

Необязательное отношение допускает ноль или одного предшественника. В канонической форме JSON нулевой случай представлен значением `null` для поля предшественника. А в реляционной таблице столбец внешнего ключа допускает NULL. Внешние ключи, которые могут быть `null`, являются причиной для беспокойства при проектировании реляционной базы данных. Типы запросов, которые мы будем выполнять, однако, как правило, позволяют избежать этих проблем.

В некоторых системах реляционных баз данных индекс на столбце, допускающем пустые значения (nullable), будет пропускать строки, в которых столбец равен `null`. В результате запросы на значения NULL будут выполнять полное

сканирование таблицы. К счастью, те формы запросов, которые мы будем строить, будут соединяться с непустыми (non-null) значениями. Индексы на внешних ключах-предшественниках, даже если это null, всегда будут использоваться в этих типах запросов.

## Много предшественников

Хранение внешнего ключа непосредственно в таблице фактов подходит для кардинальности один (*one*) или необязательно (*optional*). Это не работает для кардинальности много (*many*). Отношения со многими предшественниками требуют ассоциативной таблицы. По традиции я называю эти ассоциативные таблицы *таблицами предшественников*. Причина в том, что они концептуально хранят ссылки предшественников данного факта. Они получили свое название от факта, который их объявляет.

Для примера рассмотрим цену продукта. Это свойство может меняться с течением времени. Каждое отдельное изменение цены регистрируется как факт «Цена» (`Price`). В соответствии с шаблоном *Mutable Property* этот факт имеет много предшественников.

```
fact Price {
  product: Product
  value: decimal
  prior: Price*
}
```

Таблица цен строится из этого факта с использованием уже рассмотренных шаблонов. Таблица имеет идентификатор и хеш. Она также имеет столбец для каждого из полей и *one* или *optional* предшественников. Однако она не имеет столбцов для многих предшественников.

```
CREATE TABLE price (
  price_id SERIAL PRIMARY KEY,
  price_hash VARCHAR(88) NOT NULL UNIQUE,
  product_id INT NOT NULL REFERENCES product,
  value DECIMAL(10, 2) NOT NULL
);

CREATE INDEX price_product
  ON price (product_id);
```

Отношение *many* представлено с помощью ассоциативной таблицы. Имя таблицы происходит от факта-преемника, определяющего отношение. Таблица содержит только внешние ключи строк предшественника и преемника, ни один из которых не может быть нулевым. Пара внешних ключей уникальна, что отражает то, что коллекции предшественников являются множествами, а не списками, они не могут содержать одного и того же предшественника для любого данного преемника. Оба внешних ключа независимо индексируются.

```
CREATE TABLE price_predecessor (
  price_id INT NOT NULL REFERENCES price,
  prior_price_id INT NOT NULL REFERENCES price,
  UNIQUE (price_id, prior_price_id)
);

CREATE INDEX price_predecessor_price
  ON price_predecessor (price_id);

CREATE INDEX price_predecessor_prior_price
  ON price_predecessor (prior_price_id);
```

И теперь должно быть немного понятнее, почему мы не добавляем ограничения уникальности на поля факта. Если бы мы определили такие ограничения для таблицы, которая имеет связанную с ней *таблицу-предшественник*, то предшественники не были бы включены в уникальный индекс. Если бы мы определили ограничение уникальности на поля, было бы невозможно вставить две цены с одинаковым значением и идентификатором продукта, даже если бы у них были разные наборы предшественников. Именно это и происходит, когда цена возвращается к своему прежнему значению.

Предшественники факта являются частью его идентичности. Но ссылки на предшественников хранятся в другой таблице. Ограничения уникальности не могут пересекать границы таблиц. Поэтому мы не будем ограничивать уникальность полей вообще. Ограничение уникальности хеша прекрасно решает эту проблему.

### Канонический хеш множества

Хеш факта, имеющего много предшественников, основывается на всех хешах этих предшественников. Каноническая форма, которую я рекомендую, представляет множество предшественников в виде массива JSON. Если предшественников нет, массив будет пустым, а не null. Если предшественник один, массив содержит один объект ссылки. Ссылочные объекты сортируются в алфавитно-цифровом порядке по хешу в кодировке base-64, так что существует только одно каноническое представление множества. Например, первая цена товара будет определяться хешем следующего документа JSON с удаленными пробелами:

```
{
  «prior»: [],
  «product»: {
  «ref»: «fK02Oge...5GFw==«
  },
  «value»: 256.98
}
```

Последующая цена будет получать хеш документа JSON с одной предыдущей ссылкой:

```
{
  «prior»: [
     {
       «ref»: «ZlUYAZV...ZQZA==«
     }
  ],
  «product»: {
       «ref»: «fK02Oge...5GFw==«
  },
  «value»: 220.98
}
```

## Вставка многих предшественников

Факт с пустым набором предшественников может быть вставлен точно так же, как мы делали это раньше. Эта строка представляет собой полный факт без связанных строк в таблице предшественников.

```
INSERT INTO price
  (price_hash, product_id, value)
  SELECT 'ZlUYAZV...ZQZA==', product_id, 256.98
     FROM product
     WHERE product_hash='fK02Oge...5GFw=='
  ON CONFLICT DO NOTHING;
```

Однако DML должен быть расширен при вставке факта с одним или несколькими предшественниками в наборе. Строки вставляются в таблицы-предшественники одновременно с таблицей-преемником. Вставки выполняются в рамках транзакции базы данных, чтобы избежать возможности использования в запросе частично записанного факта.

```
BEGIN TRANSACTION;

INSERT INTO price
  (price_hash, product_id, value)
  SELECT 'DRFoFgF...BNpw==', product_id, 220.98
       FROM product
       WHERE product_hash='fK02Oge...5GFw=='
  ON CONFLICT DO NOTHING;
```

```
INSERT INTO price_predecessor
  (price_id, prior_price_id)
  SELECT successor.price_id, predecessor.price_id
     FROM price as successor, price as predecessor
     WHERE successor.price_hash = 'DRFoFgF...BNpw=='
       AND predecessor.price_hash IN ('ZlUYAZV...ZQZA==')
  ON CONFLICT DO NOTHING;

COMMIT TRANSACTION;
```

Используя предложение `IN` в операторе вставки предшественника, мы можем перечислить хеши всех предшественников, которых хотим вставить. База данных создаст все первичные ключи в одном операторе.

В данном конкретном примере таблицы предшественников и преемников имеют одинаковый тип. По этой причине в операторе вставки необходимо создать псевдонимы, чтобы разграничить строки предшественника и преемника. Это не всегда необходимо, но вы можете захотеть всегда использовать эти псевдонимы для обеспечения согласованности.

## Запросы

Мы преобразовали спецификации фактов в ограниченные операторы языка определения данных и языка манипулирования данными. Теперь мы преобразуем конвейеры в язык запросов Data Query Language. При создании запроса вам будет гораздо проще начать с конвейера, чем с определения Factual-запроса. Половина работы уже сделана. Мы просто превращаем каждый шаг конвейера в соединение, а каждый фильтр – в подзапрос.

У каждого запроса есть начальная точка. Мы выражаем начальную точку через ее хеш. Идентификатор никогда не покидал базу данных, поэтому мы не можем начать запрос с идентификатора. Начальная форма запроса выбирает начальный тип `FROM` и использует хеш в предложении `WHERE`.

```
SELECT reference_number
FROM catalog
WHERE catalog_hash='8rC0hVD...ifdw==';
```

Первоначальный запрос почти никогда не используется в своей основной форме. Любое приложение, которое может предоставить хеш, вероятно, уже имеет поля. Вместо этого мы начинаем с этого начального запроса и добавляем шаги. Каждый шаг становится SQL-соединением.

### Соединения

Каждый шаг в конвейере следует за ролью вверх к предшественнику или вниз к преемнику. Роль имеет кардинальность *one*, *optional* или *many*. Шаг, следующий за ролью с *one* или *optional*, становится простым соединением по внешнему ключу. Если это шаг вверх к предшественнику, то внешний ключ находится в текущей таблице. Если это шаг вниз к преемнику, как в следующем запросе, тогда внешний ключ находится в удаленной таблице:

```
SELECT product_hash, sku
FROM catalog
JOIN product ON catalog.catalog_id = product.catalog_id
WHERE catalog_hash='8rC0hVD...ifdw==';
```

Для шагов, следующих за ролью предшественника *many*, нам необходимо присоединиться через *таблицу предшественников*. Мы увидим пример этого, когда перейдем к подзапросу.

## Коррелированные подзапросы

Когда мы обрабатываем фильтр в конвейере, мы преобразуем его в подзапрос. В частности, он становится *коррелированным подзапросом*, поскольку зависит от значений текущей строки. В фильтре используется квантификатор `exists` или `not exists`. Он переводится непосредственно в предложение SQL `EXISTS` или `NOT EXISTS`.

Предположим, мы хотим выбрать текущую цену для продукта. Конвейер для этого запроса показан на рис. 10.1.

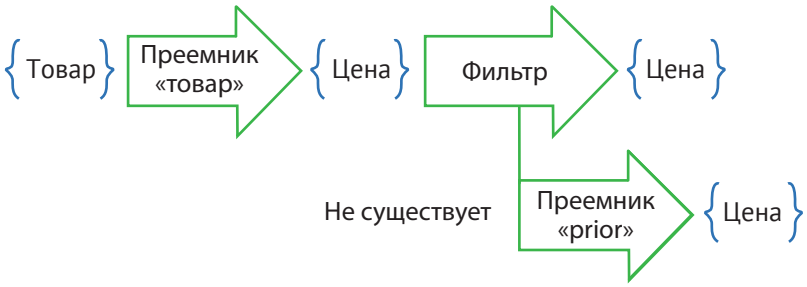


**Рис. 10.1.** Конвейер для текущей цены продукта фильтрует цены на основе подзапроса

Начнем с продукта. Определите предложение `WHERE` на основе его хеша. Затем мы пройдем до цен преемников для этого продукта.

```
SELECT price_hash, value
FROM product
JOIN price ON price.product_id = product.product_id
WHERE product_hash = 'fK02Oge...5GFw==';
```

На этом этапе конвейер отправляет нас в фильтр. Подзапрос начинается с цены. Он переходит к цене-преемнику, используя роль `prior`. Это становится соединением с использованием таблицы-предшественника `price_predecessor`.

```
SELECT price_hash, value
FROM product
JOIN price ON price.product_id = product.product_id
WHERE product_hash = 'fK02Oge...5GFw=='
  AND NOT EXISTS (
     SELECT 1
     FROM price_predecessor
     WHERE price_predecessor.prior_price_id = price.price_id
  );
```

Этот запрос пропустит цены, которые были заменены. Он вернет только текущие цены. Фильтр в конвейере становится коррелированным подзапросом, содержащим все шаги подчиненного конвейера.

## Производные таблицы

При вычислении инверсии мы обращаемся к подзапросам рекурсивно. Самый внутренний подзапрос становится начальной точкой инверсного конвейера. Мы каскадируем результаты этого конвейера в другой конвейер. Один из примеров показан на рис. 10.2.

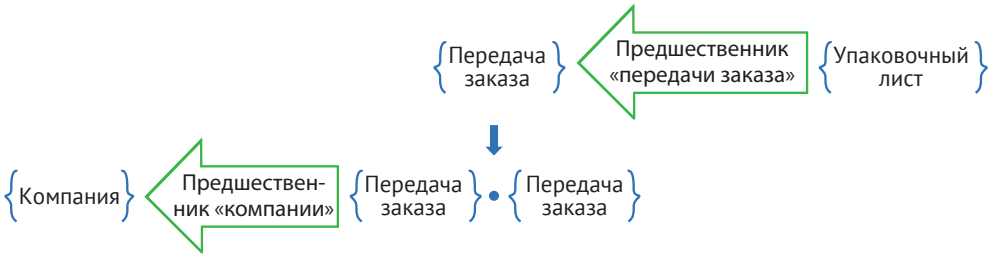


**Рис. 10.2.** Набор, произведенный верхним конвейером, поступает в нижележащий конвейер

Чтобы представить каскадный набор в SQL-запросе, примем исходный конвейер в качестве подзапроса. Однако, в отличие от фильтров, это не коррелированный подзапрос, а *производная таблица*. В то время как коррелированные подзапросы появляются в предложении `WHERE`, производные таблицы появляются в предложении `FROM`. Оператор SQL для реализации конвейера на рис. 10.2 должен выбрать передачу заказа в качестве производной таблицы следующим образом:

```
SELECT company_hash, order_submission_hash
FROM (
  SELECT company_id, order_submission_hash
  FROM packing_slip p
  JOIN order_submission os
       ON p.order_submission_id = os.order_submission_id
  WHERE packing_slip_hash = 'xxxxx...yyyy=='
) AS sub
JOIN company
  ON sub.company_id = company.company_id
```

В этом примере подзапрос относится к начальному конвейеру – упаковочный лист в передачу заказа. Он содержит хеш начальной точки. Подзапрос возвращает внешние ключи и хеши, которые будут использоваться во внешнем запросе. Слева от точки конвейер вычисляет затронутый набор. В данном случае он делает шаг к предшественнику компании. Это преобразуется в соединение с компанией. Справа от точки конвейер вычисляет новые результаты. В данном примере в представление добавляется сама передача заказа. И поэтому подзапрос должен вернуть достаточно информации для материализации этих объектов.

## Выбор результатов

Все запросы, которые мы изучали до сих пор, выбирают несколько столбцов из таблицы результатов. В общем случае вам может понадобиться и немного больше информации. Чтобы преобразовать результаты в объекты, которые может использовать ваше приложение, вы можете обнаружить, что вам необходимо получить информацию от предшественников. Для этого вам понадобятся дополнительные соединения.

Все соединения, полученные из шагов, будут *внутренними соединениями*. То есть в запрос будут включены только результаты, для которых существуют обе стороны отношения. Внешние соединения или, в частности, *левые соединения* (*left joins*) будут использоваться только для загрузки предшественников результирующего факта. Если эти предшественники имеют кардинальность *optional* или *many*, то левое соединение необходимо для того, чтобы результаты были полными.

В примере, который мы только что рассмотрели, одних хешей недостаточно для создания новых результатов. Кроме того, нам необходимо включить информацию о заказанных товарах. Эта информация доступна в таблицах-предшественниках. Чтобы получить эту информацию, мы выполним левое внешнее соединение.

```
SELECT company_hash, order_submission_hash, order_line_product_hash
FROM (
  SELECT company_id, order_submission_hash, order_submission_id
  FROM packing_slip p
  JOIN order_submission os
       ON p.order_submission_id = os.order_submission_id
  WHERE packing_slip_hash = 'xxxxx...yyyy=='
) AS sub
JOIN company
  ON sub.company_id = company.company_id
LEFT JOIN order_submission_predecessor
  ON sub.order_submission_id = order_submission_predecessor.order_
  submission_id
LEFT JOIN order_submission_line
  ON order_submission_predecessor.order_submission_line_id =
       order_submission_line.order_submission_line_id
```

В данном примере предшественники имеют кардинальность *many*. По этой причине мы представили эту связь с помощью таблицы предшественников. Возможно, что результат может не иметь предшественников. Если бы это было так и мы использовали внутреннее объединение, результат был бы исключен. Но, используя левое внешнее соединение, мы гарантируем, что результат появится, даже если у него нет предшественников.

## Оптимизация

Из приведенных ранее примеров вы должны оценить богатство и сложность языка DQL, который может быть порожден конвейером. Но у вас могут возникнуть некоторые опасения по поводу менее идеальных конструкций, которые он иногда генерирует. В предыдущих примерах мы переводили данные непосредственно из конвейеров в SQL. Теперь оптимизируем этот SQL, чтобы реализовать тот же конвейер, но более эффективным способом.

Когда мы переводили инвертированный запрос в производную таблицу, мы в итоге создали SQL, который был излишне вложенным. Он включал производную таблицу, которая впоследствии соединялась с другими таблицами. Вместо того чтобы использовать производную таблицу, мы могли бы просто объединить соединения подзапросов с соединениями основного запроса. Не все инверсии будут создавать производные таблицы, которые можно оптимизировать, но самые простые инверсии обычно оптимизируются. Переработка SQL позволяет получить код, который легче понять и поддерживать.

```
SELECT c.hash
FROM (
  SELECT b_id
  FROM a
  JOIN b ON b.a_id = a.a_id
  WHERE a.hash = 'a_hash'
) AS sub
JOIN c ON c.b_id = sub.b_id
```

```
SELECT c.hash
FROM a
JOIN b ON b.a_id = a.a_id
JOIN c ON c.b_id = b.b_id
WHERE a.hash = 'a_hash'
```

Хотя эта форма оптимизации делает код лучше, механизм запросов будет создавать точно такой же план для любого оператора SQL. Если вы пишете этот код вручную, то имеет смысл его оптимизировать. Но вы можете счесть более продуктивным просто позволить генератору кода производить DQL из конвейера. В этом случае пусть генератор кода производит код, который легче всего сгенерировать, а не тот, который легче всего понять. Производительность во время выполнения будет точно такой же.

## Ложные соединения

Одна из проблем, которую немного сложнее оптимизировать для системы запросов, – это соединение, которое не нужно. Такие ложные соединения часто возникают, когда конвейер делает шаг к предшественнику, а затем сразу же спускается к другому преемнику. Например, чтобы найти имя учащегося, зачисленного в класс, мы можем использовать конвейер, показанный на рис. 10.3.


**Рис. 10.3.** Поиск имени зарегистрированного учащегося включает в себя шаг к учащемуся (student), а затем к имени (student name)

Прямой перевод этого конвейера на язык SQL даст запрос с двумя соединениями:

```
SELECT name
FROM registration
JOIN student
  ON registration.student_id = student.student_id
JOIN student_name
  ON student.student_id = student_name.student_id
WHERE registration_hash = 'xxxxx...yyyy=='
```

Однако при внимательном рассмотрении выясняется, что соединение с таблицей учащихся не дает никакой дополнительной информации. Идентификатор учащегося уже известен. Поэтому мы можем убрать промежуточное соединение.

```
SELECT name
FROM registration
JOIN student_name
  ON registration.student_id = student_name.student_id
WHERE registration_hash = 'xxxxx...yyyy=='
```

Соединение напрямую от одного потомка к другому без прохождения родителя возможно, потому что нам не нужна дополнительная информация. Однако если требуется какое-то поле родителя, то такая оптимизация нам недоступна. Генератор SQL может искать эту возможность и создать оптимизированный запрос. Такая оптимизация даст более эффективный план запроса.

## Охватывающие индексы

Когда вы анализируете планы запросов, созданные этими типами запросов, в них доминируют поиск индекса и сканирование индексов. Поиск по индексу – это самый эффективный способ, которым движок базы данных может реализовать объединение. Для каждого ключа во входном наборе движок ищет определенное место в индексе, чтобы найти соответствующий ключ в выходном наборе. Сканирование индексов аналогично, но используется для соединений, которые возвращают много связанных строк – соединения преемников или соединения *многих* предшественников. Мы можем воспользоваться преимуществами этих быстрых операций запроса, потому что создали индексы для всех внешних ключей.

Однако в конце плана запроса обычно встречается поиск *кластеризованного индекса*. Запрос создал набор первичных ключей с помощью различных операций поиска и сканирования индексов. Теперь он использует эти первичные ключи для поиска строк в таблице с целью извлечения значений в предложении SELECT. Эта последняя операция может занять значительную часть времени выполнения запроса.

Для ускорения запросов, в которых выбираются дополнительные столбцы, иногда целесообразно включить эти выбранные столбцы в *охватывающий индекс*. При создании индекса, основанного на внешнем ключе, можно включить столбцы данных. Значения из этих столбцов будут скопированы в индекс и станут легкодоступными.

```
CREATE INDEX price_product
  ON price (product_id)
  INCLUDE (value);
```

Хотя это может улучшить производительность часто используемых запросов, охватывающие индексы влекут за собой определенные расходы. Поскольку индекс включает дополнительные данные, он будет занимать больше места. Вставка индекса и работа с ним займет немного больше времени. Это займет меньше времени, чем поиск индекса с последующим поиском кластеризованного индекса, но разницу в производительности не следует игнорировать. Охватывающие индексы нужно использовать редко и только после профилирования запросов.

## Where Not Exists

Наиболее существенная проблема, связанная с производительностью, возникает из-за экзистенциальных условий. Многие из запросов, которые мы создаем в реальных приложениях, включают в себя условие `WHERE NOT EXISTS`. Чаще всего это условие возникает в шаблонах *Delete*, *Mutable Property* и *Queue*. Оно приводит к антисоединению (anti join). Это сканирование индекса, возвращающее только входные строки, для которых не создается выходная строка. Сама по себе эта операция не является медленной, особенно потому, что она может использовать индекс внешнего ключа. Проблема возникает, когда она выполняется над большим входным набором, даже если в результате получается небольшой выходной набор.

Сканирование индекса будет выполняться для каждой входной строки. Производительность не связана с количеством выходных строк. Это означает, что проблемы с производительностью могут скрываться в запросах, которые кажутся небольшими. Основываясь на том, как используется шаблон, мы можем оценить, сколько ключей будет отсканировано, по сравнению с тем, сколько результатов будет получено. Вы можете определить процент *отходов* – это отношение количества исключенных входов к общему количеству входов. Для того чтобы производительность примерно коррелировала с объемом, мы хотели бы, чтобы отходы были ограниченными и достаточно низкими.

### Изменяемые свойства

Если в запросе на текущее значение изменяемого свойства встречается условие `WHERE NOT EXISTS`, то входной набор будет включать все исторические значения. Выходной набор будет содержать только те версии, которые не были заменены. Таким образом, отходом будет отношение вытесненных версий к общему количеству версий. Для изменяемого свойства, которое меняется часто, отходы будут высокими. Но если свойство меняется относительно редко, отходы будут низкими.

По этой причине шаблон Mutable Property следует применять только к медленно изменяющимся свойствам. Вещи, которые часто меняются в течение жизни сущности, производят значительные отходы. Медленно изменяющиеся свойства – это, например, имя человека или адрес компании. Часто меняющиеся свойства – это состояние или продвижение. Шаблоны рабочего процесса нередко являются лучшей альтернативой для постоянно меняющихся свойств.

### Удаление

Еще одно место, где то и дело появляется `WHERE NOT EXISTS`, – это запрос, который исключает удаленные сущности. Входной набор включает все сущности, которые когда-либо существовали. Выходной набор будет содержать только те, которые все еще существуют. Отходом является отношение числа удаленных сущностей к общему числу сущностей. Когда большинство сущностей удаляется, отходы высоки.

Вспомните последнее CRUD-приложение, которое вы создали. Вероятно, там была одна большая таблица, включающая основную сущность, с которой работало приложение. Возможно, это была таблица клиентов, которую вы часто запрашивали, чтобы получить большой список текущих клиентов. Поскольку клиенты приходят и уходят, со временем удалялось все больше и больше клиентов по сравнению с относительно стабильным количеством текущих клиентов.

Если вы использовали реальный оператор `DELETE`, то отходы были ограничены. Но если вы использовали мягкое удаление, то отходы со временем росли. При мягком удалении вы устанавливали специальное поле для имитации удаления, а затем выбирали только те строки, в которых флаг не установлен. Мягкое удаление сохраняло данные, но это было сопряжено с расходами на производительность.

Удаление в исторической модели аналогично мягкому удалению в CRUD-модели. Оно сохраняет данные, но несет те же затраты на производитель-

ность. К счастью, у нас есть несколько вариантов восстановления производительности.

Первый вариант – добавить в модель часы. См. шаблон Period в главе 8. Часы – это отправная точка для запроса, который медленно продвигается во времени. Некоторые модели имеют естественные периоды, например даты начала работы в системе розничной торговли или семестры в академической области. В других моделях часы более искусственные. В любом случае запрос, основанный на небольшом количестве периодов, ограничивает количество потерь из-за удаления. Только выжившие переходят из одного периода в другой.

Если не удается найти разумные часы, то можно прибегнуть к более радикальному решению. Измерьте свою производительность и рассчитайте фактический процент отходов. Если вы обнаружите, что большинство сущностей удаляется и это влияет на производительность запросов с течением времени, тогда подумайте об управляемом индексе. Это решение наиболее применимо к очередям, поэтому мы рассмотрим его в данном контексте.

## Очереди

Шаблон Queue, как описано в главе 8, включает факт-преемник, который записывает результат выполнения действий над рабочим элементом. Очередь питается запросом, который выбирает рабочие элементы, для которых не существует результата. Они преобразуются в фильтры конвейера, которые в конечном итоге превращаются в условия `WHERE NOT EXISTS` в SQL. Входным набором для антисоединения является множество всех рабочих элементов. Выходное множество содержит только те рабочие элементы, которые еще не обработаны. Поскольку каждый рабочий элемент в конечном итоге будет обработан, этот запрос приближается к 100%-му отходу.

Полагаясь на антисоединение, мы лишаем SQL-движок самого ценного инструмента – прямой адресации. Элементы в очереди не являются непосредственно адресуемыми. Вместо этого индекс хранит именно те элементы, которых *нет* в очереди. Движку приходится сканировать этот индекс для каждого рабочего элемента, чтобы увидеть, какие из них не исключены. Это было бы намного быстрее, если бы мы смогли убедить SQL-движок напрямую индексировать те элементы, которые были включены.

Чтобы убедить движок, мы должны предоставить ему что-то, что он может непосредственно индексировать, и обратиться к нему. Что-то, что представляет собой *отсутствие* преемника. Один из вариантов – пересмотреть флаг мягкого удаления.

Мягкое удаление, реализованное в CRUD-системе, представляет собой обновление (`UPDATE`), которое заменяет собой удаление (`DELETE`). Вместо того чтобы полностью уничтожить запись, приложение обновляет ее путем установки флага. Такой флаг может быть проиндексирован, чтобы обеспечить прямой доступ к тем записям, которые не были помечены как удаленные. Когда относительно небольшое количество строк помечено, индекс не является избирательным. Большинство движков запросов будут избегать неселективного индекса,

потому что сканирование таблицы окажется быстрее. Но если большинство строк помечены, как в рабочих элементах, обрабатываемых из очереди, тогда индекс становится достаточно селективным. Движок запросов может использовать индекс для прямого обращения к относительно небольшому числу необработанных рабочих элементов.

Даже при избирательном подходе индекс с флагом имеет недостаток пространства. Все строки индексируются, даже те, для которых установлен флаг. Индекс никогда не будет использован для поиска отмеченных рабочих элементов, потому что запрос специально ищет неотмеченные. Мы можем сэкономить немного места, если полностью исключим из индекса обработанные рабочие элементы.

Некоторые системы управления базами данных включают функцию *частичного индекса*. Если эта функция доступна, вы можете создать индекс с условием. Укажите это условие, чтобы включить только необработанные рабочие элементы в индекс, и данный индекс будет использоваться для прямого обращения к этим элементам.

```
CREATE INDEX work_item_unprocessed
  ON work_item (queue_id)
  WHERE processed = FALSE;
```

Но даже если ваша система управления базой данных не поддерживает эту функцию, вы можете смоделировать ее. Создайте таблицу, которую вы будете использовать в качестве индекса. Вставьте строки в эту таблицу, чтобы указать, что флаг не установлен, и удалите их, чтобы указать, что он установлен.

```
CREATE TABLE work_item_unprocessed (
  queue_id INT NOT NULL,
  work_item_id INT NOT NULL UNIQUE
)
CREATE INDEX work_item_unprocessed_queue
  ON work_item_processed (queue_id);
```

Добавляете ли вы изменяемый флаг и определяете частичный индекс, или вы добавляете таблицу для хранения необработанных рабочих элементов, вам придется самостоятельно управлять индексом. По этой причине я называю эту технику *управляемым индексом*. Я рекомендую использовать триггеры для управления индексами. Хотя индекс может управляться приложением, он существует для решения проблемы производительности базы данных. Поэтому решение должно быть полностью определено в базе данных.

Чтобы использовать подход индексной таблицы, создайте два триггера. Первый вставляет данные в индексную таблицу, когда создается новый рабочий элемент. Второй удаляет их из индексной таблицы, когда создается результат. Например, следующая пара триггеров управляет индексной таблицей для очереди:

```
CREATE FUNCTION insert_work_item_unprocessed() RETURNS TRIGGER AS $$
  BEGIN
     INSERT INTO work_item_unprocessed
        (queue_id, work_item_id)
        VALUES (NEW.queue_id, NEW.work_item_id);
     RETURN NULL;
  END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER work_item_created AFTER INSERT ON work_item
  FOR EACH ROW
  EXECUTE FUNCTION insert_work_item_unprocessed();
INSERT INTO work_item
  (queue_id, description)
  VALUES (47, 'Do some work');
CREATE FUNCTION delete_work_item_unprocessed() RETURNS TRIGGER AS $$
  BEGIN
     DELETE FROM work_item_unprocessed
       WHERE work_item_id = NEW.work_item_id;
     RETURN NULL;
  END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER work_item_outcome_created AFTER INSERT ON work_item_outcome
  FOR EACH ROW
  EXECUTE FUNCTION delete_work_item_unprocessed();
```

Присоединяйтесь через индексную таблицу вместо использования условия `WHERE NOT EXISTS`. Оптимизатор запросов будет использовать таблицу для прямого обращения к элементам, оставшимся в очереди. Ему не нужно будет выполнять сканирование всех исторических рабочих элементов, чтобы выполнить антисоединение. Результаты будут намного быстрее, особенно для рабочих элементов, 100 % которых в конечном итоге исключаются из запроса.

## Интеграция

Не все функции приложения будут обслуживаться из неизменяемой базы данных. Некоторые запросы не поддаются обходу отношений предшественников и преемников из одной исходной точки. Некоторые структуры таблиц лучше денормализовать. Когда более подходящей является изменяемая структура, основанная на состоянии, используйте эту структуру. Интегрируйтесь между неизменяемой базой данных и изменяемой.

При разделении изменяемых и неизменяемых данных убедитесь, что структуры данных изолированы друг от друга. Не смешивайте две структуры таблиц. Не включайте внешние ключи из одной модели в другую. И избегайте любой формы перекрестного соединения между двумя моделями. Даже если имеет смысл использовать один и тот же движок реляционной базы данных для двух моделей, не объединяйте их в одну базу данных. У них разные цели и модели использования.

## Интеграция унаследованных приложений

При внедрении неизменяемой архитектуры в организации у вас не будет возможности переписать все ваши системы в этом стиле. Да и не стоит искать такой возможности. Это повлечет за собой большой риск, а выгоды будут получены слишком поздно. Вместо этого позвольте унаследованным приложениям продолжать существовать, имея статические, изменяемые модели данных. Найдите лучшие точки интеграции для работы с этими унаследованными приложениями с минимальным риском и доработкой.

Некоторые унаследованные приложения могли быть написаны с учетом архитектуры, управляемой событиями. Если это так, используйте точки интеграции в качестве мест, где исторические факты производятся и объявляются. Однако более вероятно, что унаследованное приложение будет просто коллекцией нестационарных моделей поведения поверх изменяемой модели данных. Оно не будет отбрасывать события, которые могут быть превращены в факты. Не будет простого способа модифицировать исходный код приложения для создания таких точек интеграции.

В этих случаях я предлагаю использовать базу данных в качестве точки интеграции. Это не значит, что два приложения должны использовать общую базу данных – такая практика приводит к тесной связи и быстрому застою. Вместо этого применяйте инструменты, присущие базе данных, для извлечения фактов из изменений состояния. К таким инструментам относятся сканеры, триггеры и захват изменений данных.

### Сканеры

Самый прямой способ извлечения фактов из данных – это их сканирование. Сканер – это создаваемый вручную запрос, отображающий текущее состояние приложения в набор фактов, которые привели к такому состоянию.

Чтобы разработать полезный сканер, вы должны понять, какие части модели могут измениться, а какие являются неизменными. Движки реляционных баз данных не имеют многих инструментов для обеспечения или документирования неизменяемости, но имеют некоторые из них. Одним из таких инструментов является ограничение уникальности.

Когда таблица получает ограничение уникальности, разработчик документирует тот факт, что включенные столбцы образуют естественный ключ для строки. Эти столбцы представляют собой идентичность, независимую от автоинкрементного суррогатного ключа. Даже если движок базы данных не запрещает приложению изменять эти столбцы, делать это настоятельно не рекомендуется. Если разработчик базы данных потрудился над объявлением ограничения уникальности, вы можете быть уверены, что столбцы неизменяемы.

Начните с создания серии запросов, которые выбирают неизменяемые данные из исходной системы. Вы должны иметь возможность отразить эти результаты непосредственно в факты. Если эти факты уже существуют в целевой неизменяемой модели, то ничего страшного – они не будут дублироваться. Просто спроецируйте эти результаты и вставьте соответствующие факты.

Далее остаются изменяемые данные. Для их проецирования потребуется немного истории. Чтобы помочь отследить свою работу, сканер должен вести

блокнот, в котором будут записаны изменяемые значения, которые он видел в последний раз для каждой сущности. Когда он находит другие значения, он может использовать этот блокнот, чтобы найти предыдущие версии и сделать эти факты предшественниками новых фактов. Конкретные детали блокнота в значительной степени зависят от вашего приложения и выбранного технологического стека. Надо только сказать, что он должен содержать достаточно информации, чтобы полностью восстановить историческое дерево предыдущих версий.

Обычно статические базы данных проектируются со столбцами аудита. В них записывается дата вставки строки и дата последнего обновления. Сканер может воспользоваться столбцами аудита, чтобы ограничить объем работы, который он должен выполнить. Сохраните закладку в блокноте сканера отдельно от исходной базы данных или целевой неизменяемой модели. Включите предложение `WHERE`, которое ограничивает сканер строками, которые были вставлены или обновлены после закладки. Обновляйте эту закладку только после того, как убедитесь, что новые факты были сохранены.

При переносе из любой старой базы данных в неизменяемое хранилище данных сканер всегда является первым инструментом. Вы можете строить сканер слой за слоем, сначала сканируя высокоуровневые факты, не имеющие предшественников, а затем создавая сканеры для преемников ниже по графу. Постоянно развертывайте сканер по мере построения последующих слоев. Сканирование выполняется не один раз. Оно будет выполняться много раз в течение жизни проекта. Не совершайте ошибку, рассматривая это как одноразовое перемещение данных. Потратьте время на то, чтобы сделать процесс повторяемым.

## Триггеры

Триггеры – это, пожалуй, самый спорный инструмент в наборе средств моделирования данных. Возможно, они в прошлом использовались слишком часто, что привело к тому, что некоторые разработчики приложений стали их избегать и даже предостерегать от них других. Но для извлечения информации из базы данных инструменты для работы с базами данных являются наиболее подходящими, особенно если наша цель – минимизировать изменения в существующем коде.

Мы уже использовали триггеры для оптимизации запросов на рабочие элементы, которые еще не были обработаны. Мы также можем использовать их для вывода фактов из операций изменения данных. Оригинальное намерение может быть заперто в коде приложения, который мы не смеем изменять, но эффект от этого намерения можно легко наблюдать. Некоторые хорошо продуманные триггеры могут преобразовать `INSERT` в новую сущность, `UPDATE` – в изменяемое свойство, а `DELETE` – в надгробие[1] (tombstone). Они могут вставлять эти новые факты в набор промежуточных таблиц для обработки по тем же каналам, что и действия пользователя.

[1] Это так называемый объект-захоронение, который помечен как удаленный, но фактически оставшийся в БД, или объект, удаленный из директории локального хранилища, но не удаленный из БД. – *Прим. ред.*

Триггеры такого рода не могут занять место сканеров. Они будут установлены после того, как статическая модель данных проработала некоторое время. Триггер не может вернуться в прошлое и выполнить все вставки, обновления и удаления данных в прошлом. Сканер все еще требуется для ввода всех унаследованных данных.

Триггеры, если они будут внедрены, должны работать в сочетании со сканерами. В какой-то момент вы можете решить отключить сканер, одновременно включив триггеры. Делайте это только в том случае, если у вас есть достаточная уверенность в том, что триггеры уловят все возможные изменения. Например, если вы используете массовые вставки, триггеры могут не сработать. В качестве альтернативы вы можете реализовать триггеры с помощью сканера. Напишите сканер в виде серии хранимых процедур. Затем триггер просто запускает сканер всякий раз, когда обнаруживает, что информация изменяется. Это дает нам лучшее из двух миров – сканер, работающий по таймеру для сбора массовых изменений данных, и триггеры, работающие в реальном времени для обеспечения меньшей задержки.

### Захват изменений данных

Описанные ранее схемы могут быть реализованы в любой системе управления базой данных. Однако некоторые системы предоставляют более широкую поддержку. Например, SQL-сервер предоставляет специальные функции, которые можно использовать для извлечения изменений из базы данных. Наиболее мощной из них является захват изменений данных.

При настройке захвата изменений данных администратор определяет исходные таблицы. Когда операции DML над любой из этих таблиц-источников попадают в журнал транзакций, процесс захвата данных уведомляется асинхронно. Процесс вставляет запись о каждом изменении в соответствующую таблицу изменений. Приложение может периодически извлекать записи из таблиц изменений и преобразовывать их в исторические факты.

Преимущество захвата изменений данных заключается в использовании журнала транзакций для асинхронной идентификации изменений. Триггеры, с другой стороны, запускаются синхронно с командой DML. Это означает, что они потенциально могут заблокировать или даже повредить унаследованное приложение. Захват изменений данных встроен в систему управления базой данных, тогда как триггеры необходимо кодировать вручную. С другой стороны, триггеры могут быть настроены для обработки более сложных сценариев. Выбор между этими двумя вариантами отнюдь не является однозначным.

Мы рассмотрели несколько альтернатив для извлечения фактов из статической базы данных. Интеграция унаследованных приложений, конечно, должна работать в обоих направлениях. Для получения информации в унаследованное приложение обратитесь к шаблону Outbox в главе 8 или используйте методы, описанные для создания отчетов по базам данных в следующем разделе.

## Создание отчетов из баз данных

Выполнение отчетов непосредственно из неизменяемой базы данных может оказаться сложной задачей. Запросы, которые вычисляют текущее состояние объекта, обычно хорошо работают для единичных начальных точек. Они

не очень хорошо подходят для агрегирования, как это часто требуется в отчетах. Они также не подходят для группировки по изменяемым свойствам, как нередко бывает в сценариях отчетов. Для получения эффективных отчетов неизменяемая модель данных должна быть преобразована в денормализованную модель отчетности.

Чтобы построить эффективную базу данных для отчетности на основе неизменяемой модели, решите, какие факты будут отражены в статической базе данных. Не все неизменяемые записи должны быть отражены в отчетах. Возможно, вы даже можете разделить отчеты на категории и создавать отдельные проекции данных для каждой из них. Предшественники верхнего уровня попадают во все проекции отчетов, но преемники могут влиять только на некоторые из них.

После определения напишите задание, которое будет вызывать процедуру для каждого факта в исторической таблице. Это может быть запланированное задание базы данных, задание приложения cron[1] или даже управляемое триггером. Выбор зависит от вашей терпимости к задержкам и вашего понимания, что приложение должно быть вовлечено в процесс. Это задание выполняется полностью в пределах неизменяемого хранилища данных. Однако процедуры, которые оно вызывает, выполняются при создании отчетов в базе данных.

На самом деле эти процедуры должны быть хранимыми процедурами. Они принимают в качестве параметров все поля факта, который они обрабатывают. Процедура преобразует факт сущности в оператор `INSERT`, факт надгробия – в `DELETE`, а любое изменяемое свойство или факт рабочего процесса – в `UPDATE`. Затем хранимая процедура возвращает идентификатор строки, которую она только что затронула. Это помогает заданию отслеживать предшественников.

По мере того как задание вызывает процедуры для обработки фактов, оно ведет карту. Оно сопоставляет хеш каждого факта с идентификатором, который вернула процедура обработки. Затем, когда вызывается процедура для факта-преемника, оно передает сопоставленный ID вместо предшественника. Это дает хранимой процедуре всю информацию, необходимую для установки внешних ключей.

Работа по преобразованию фактов в изменения базы данных полностью выполняется внутри хранимых процедур. Это сохраняет в базе данных информацию о схеме отчетов базы данных. Лица, отвечающие за создание и оптимизацию отчетов, найдут этот инструмент под рукой. Это решение обеспечивает доступность, удобство и автономность, необходимые для создания наилучших отчетов.

## Агностичные к приложениям хранилища

На протяжении всей этой главы мы много кодировали вручную, чтобы превратить модель Factual в реляционную базу данных. Мы преобразовывали типы фактов в таблицы, предшественников в ассоциации, а конвейеры – в запросы. Понимание этого процесса было полезным, но в конце концов это не по-

[1] Cron – это хронологический демон-планировщик задач, работающий в операционных системах типа Unix, включая дистрибутивы Linux. Cron запускается в фоновом режиме, а задачи, запланированные в cron, выполняются автоматически, что делает cron полезным для автоматизации связанных с обслуживанием задач. – *Прим. перев.*

требовало никакого творчества. Он оказался довольно механическим. Если мы полностью примем автоматизацию, то сможем генерировать код, упростить развертывание базы данных и избежать многих головных болей, связанных с версионированием.

Неизменяемое хранилище данных накладывает значительный набор ограничений. Одно из них – это, конечно, то, что строки не могут быть обновлены или удалены. Но при создании структуры неизменяемой базы данных из конвейеров и инверсий мы получили несколько других ограничений. Предложение `WHERE` выбирает первый ряд по хешу. Все соединения выполняются по предшественнику или по внешним ключам-преемникам. Внешние соединения используются только для конечных предшественников. Тем не менее эти ограничения помогают нам писать эффективные и корректные запросы.

В частности, эти ограничения не позволяют нам запрашивать базу данных по любому из полей данных. Мы не можем найти все товары по определенной цене или все грузы свыше определенного веса. Такие запросы не могут быть эффективно написаны для неизменяемой базы данных и вместо этого должны выполняться в статической проекции, например в базе данных отчетов. При подобных ограничениях можно задуматься, зачем вообще хранить цену и вес в столбцах.

Базы данных, поддерживающие специальные запросы, такие как базы данных отчетов, выигрывают от возможности определить схему, специфичную для конкретного приложения. Если в отчете необходимо фильтровать или группировать по какому-то полю, то для этого поля нужен столбец. Однако конвейеры запрещают нам выполнять специальные запросы. Единственная причина иметь столбцы данных – это возможность хранить и извлекать поля. Если эта цель может быть достигнута другим способом, то все таблицы начинают приближаться к одной и той же форме.

## Общая таблица фактов

Большинство реляционных систем баз данных имеют эффективный способ хранения коллекций именованных значений. Например, в PostgreSQL объект JSON можно хранить в двоичной форме, что позволяет сократить расходы на кавычки и экранирование строк. Это делается в первую очередь для того, чтобы индексировать поля объекта JSON, но нам не нужно использовать его таким образом. Мы можем просто хранить и извлекать целые объекты. И даже если выбранный вами движок базы данных не имеет двоичного типа JSON, вы можете легко хранить объекты JSON в виде строк.

Как только вы начнете хранить поля в столбцах JSON, а не в индивидуальных столбцах, различия между таблицами, предназначенными для конкретных приложений, начинают исчезать. Все они имеют автоинкрементный суррогатный ключ и хеш соответствующего факта. Последняя оставшаяся причина для наличия разных таблиц – это различие между фактами разных типов.

Даже тип может быть обобщен. Мы можем однозначно определить тип факта по его имени и хранить его в таблице типов, чтобы дать ему суррогатный ключ. Затем можем хранить факт ID, ID типа, хеш факта и поля JSON в таблице фактов. Благодаря такой структуре, не зависящей от приложений, мы сохранили всю важную информацию.

```
CREATE TABLE type (
  type_id SERIAL PRIMARY KEY,
  name VARCHAR(50) NOT NULL UNIQUE
)
CREATE TABLE fact (
  fact_id SERIAL PRIMARY KEY,
  type_id INT NOT NULL REFERENCES type,
  fact_hash VARCHAR(88) NOT NULL,
  fields JSONB NOT NULL,
  UNIQUE (fact_hash, type_id)
);
```

Уникальный индекс объединяет хеш с типом. Если этого не сделать, то два факта с одинаковыми полями, но разным типом будут конфликтовать.

## Отношения предшественников

Когда все факты хранятся в одной таблице, все отношения предшественников могут быть сгруппированы вместе. В неизменяемой базе данных, специфичной для конкретного приложения, единичные предшественники представлены в виде внешних ключей в таблице преемников. Только множественные предшественники были извлечены в ассоциативные таблицы. Но в хранилище данных, не зависящем от приложения, мы будем извлекать все отношения предшественников.

Подобно тому, как каждый факт имеет тип, каждое отношение предшественника имеет роль. Роль – это именованное отношение, определенное в типе факта. Например, в типе факта «Продукт» (`Product`) каталог (`catalog`) – это роль.

```
fact Product {
  catalog: Catalog
  sku: string
}
```

Каждая роль объявлена одним типом и нацелена на другой. Роль `catalog` объявлена типом `Product` и нацелена на `Catalog`. Для уникальной идентификации роли требуется как объявляющий тип, так и имя уникальной идентификации роли. Мы фиксируем это в таблице, чтобы дать роли суррогатный ключ.

```
CREATE TABLE role (
  role_id SERIAL PRIMARY KEY,
  declaring_type_id INT NOT NULL REFERENCES type,
  target_type_id INT NOT NULL REFERENCES type,
  name VARCHAR(50) NOT NULL,
  UNIQUE (declaring_type_id, name)
);
```

Идентификатор роли позволяет определить таблицу для каждого отношения-предшественника. Здесь я назову эту таблицу `edge`, чтобы указать, что это ребро в направленном ациклическом графе. Ребро соединяет одного преемни-

ка с одним предшественником в рамках роли. Все три столбца вместе образуют уникальный ключ.

```
CREATE TABLE edge (
  predecessor_id INT NOT NULL,
  successor_id INT NOT NULL,
  role_id INT NOT NULL,
  UNIQUE (successor_id, role_id, predecessor_id)
);

CREATE INDEX edge_predecessor
  ON edge (predecessor_id, role_id);
```

Уникальный индекс, определенный ранее, удваивается как способ перехода от преемника к предшественнику. Используя префикс индекса, запрос может присоединиться к известному идентификатору преемника и роли, чтобы найти предшественника. Этот индекс, однако, будет бесполезен в другом направлении. Поэтому мы определили второй индекс, начинающийся с идентификатора предшественника при соединении вниз по графу.

С помощью этих четырех таблиц – типов, фактов, ролей и ребер – мы можем хранить любую неизменяемую модель данных. Используя те же методы, которые мы изучали в предыдущих разделах, мы можем генерировать DQL непосредственно из конвейеров для такого хранилища данных, не зависящего от приложений. Более того, генерация запросов может быть автоматизирована. Вы вряд ли захотите писать очень много таких запросов вручную, но машина может сделать это с легкостью. Библиотека с открытым исходным кодом Jinaga является лишь одним из примеров системы, которая разбирает спецификации на конвейеры, вычисляет инверсии и генерирует SQL-запросы от имени приложения.

Преимущества хранилищ данных, не зависящих от приложений, начинаются с генерации кода, но они идут значительно дальше. Гораздо легче развернуть единую схему базы данных, которая не будет меняться по мере добавления функций в приложение. Многие приложения могут совместно использовать одно и то же хранилище данных, что снижает общую стоимость операций. Легче перемещать данные из одного хранилища в другое, если вы можете быть уверены, что они имеют одну и ту же схему. Но самое большое преимущество – это то, что хранилища данных, не зависящие от приложений, решают проблему управления версиями.

## Управление версиями

В любую базу данных, предназначенную для конкретного приложения, встроено скрытое ограничение неизменяемости. По мере того как приложение развивается, неизбежно, что схема будет развиваться вместе с ним. Неизменяемость требует, чтобы факт не менялся. Как мы уже много раз говорили, это означает, что не должно быть никаких «обновить» (`UPDATE`) и «удалить» (`DELETE`). Но это также означает отсутствие изменений таблиц (`ALTER TABLE`). Это не только значения, которые не могут меняться, но и форма.

При расширении неизменяемой структуры данных лучше всего определять совершенно новые типы фактов. Не следует добавлять поля или предшест-

венников к существующему типу фактов. Причина проста – существующие факты не имеют таких полей или предшественников. Их добавление означает изменить неизменяемый факт. Любая модификация изменяет идентичность факта. Все математические и аналитические структуры, которые мы тщательно собирали в течение всей этой книги, рассыпаются на песке меняющихся сущностей.

К счастью, большинство расширений, которые вы можете захотеть сделать в течение жизни приложения, имеют форму новых преемников. Вы можете захотеть добавить новое свойство к сущности. Это прекрасно – изменяемые свойства моделируются как факты-преемники. Вам может понадобиться добавить еще один шаг в рабочий процесс. Легко! Каждый шаг – это факт-преемник. Добавление нового преемника к существующей модели не является разрушающим изменением. Это именно тот вид расширения, который хорошо воспринимают неизменяемые модели.

Но иногда вы обнаруживаете, что вам действительно нужно изменить форму типа факта. Вы можете обнаружить, что факт верхнего уровня должен быть вложен под ранее невидимым владельцем. Или вы могли подумать, что одного поля достаточно для уникальной идентификации сущности, а позже обнаружили, что их нужно два. В этих случаях вы действительно намереваетесь расширить идентичность факта путем добавления предшественников или полей. Как вы можете сделать это без модификации существующих фактов? Ответ – версионирование. Существующие факты сохраняют форму, в которой они были впервые определены. При повторном извлечении этих фактов из хранилища данных мы не хотим устанавливать новых предшественников на null или определять значения по умолчанию для новых полей. Мы по-прежнему хотим загрузить старые факты, используя старую схему. Это единственный способ сохранить их идентичность.

### Избегайте последовательных номеров версий

Обычной практикой является присвоение типам последовательных номеров версий. Это можно увидеть в API, которые часто сохраняют старые версии конечных точек, чтобы они не ломали старые версии клиентов. Одна версия конечной точки принимает и возвращает старые версии структур данных, а более новая версия той же конечной точки принимает и возвращает новые структуры данных. Даже принято использовать порядковый номер версии в URL-адресе конечной точки.

Однако в неизменяемой структуре данных последовательных номеров следует избегать. Просто, как и в случае с автоинкрементными идентификаторами, порядковый номер зависит от местоположения. Он подразумевает, что все версии возникли в одном и том же месте. В случае первичного ключа в базе данных таким местом является сервер базы данных. С последовательным номером версии это место – разработчик.

На первый взгляд, кажется разумным предположить, что все версии типа факта будут исходить от одного и того же разработчика, что они будут появляться одна за другой в предсказуемом порядке. Но на самом деле разработчики работают в командах. Они не обязательно заканчивают работу в том же порядке, в котором они ее начали. Они могут даже не развертывать работу в том

же порядке, в котором они ее завершают. Порядок появления версии может отличаться от порядка присвоения номера.

К счастью, проблема версионирования может быть решена точно так же, как и проблема идентичности. Версия типа факта определяется его формой. Отдельный набор полей и предшественников, определяемых этим типом, является версией типа. А версия может быть представлена хешем.

### Структурное версионирование

Следуя алгоритму, похожему на вычисление хеша факта, мы сначала определяем каноническую форму типа факта. Начните с имени типа. Отсортируйте поля и предшественников в алфавитном порядке по имени и включите описание их типа. Для полей это будет их родной тип, а для предшественников – их кардинальность и имя целевого типа. Сериализуйте каноническую форму и вычислите хеш.

```
$ echo -n 'Price{prior:Price*;product:Product;value:decimal}' | \
> openssl dgst -sha512 -binary | \
> base64
8/HZgkV...lZMw==
```

Версия хранится вместе с каждым фактом. Чтобы оптимизировать хранение, вы можете вставить новую таблицу версий между фактом и типом.

```
CREATE TABLE version (
  version_id SERIAL PRIMARY KEY,
  type_id INT NOT NULL REFERENCES type,
  version_hash VARCHAR(88) NOT NULL UNIQUE
);
CREATE TABLE fact (
  fact_id SERIAL PRIMARY KEY,
  version_id INT NOT NULL REFERENCES version,
  fact_hash VARCHAR(88) NOT NULL UNIQUE,
  fields JSONB NOT NULL
);
```

Факт был создан с определенной версией типа. Он должен быть повторно создан с той же версией. Встройте каждую из версий в приложение, чтобы оно могло загрузить факт в соответствующую структуру данных. Если приложение получает хеш версии, которую оно не распознает, то оно просто игнорирует этот факт *и его преемников*. Он должен быть создан в более поздней версии приложения, и поэтому весь поднабор графа представляет данные, которые он не может правильно интерпретировать.

Роли не должны использовать определенные версии фактов. Отношения предшественник/преемник между фактами, которые мы используем в запросе, будут сохраняться от версии к версии. Новые версии должны иметь возможность участвовать в этих отношениях наряду со старыми версиями. Если мы включим идентификатор версии в роль, то новые версии преемников могут быть связаны только с новыми версиями предшественников. Как только мы

начинаем тянуть за эту нить, гобелен непрерывности приложения быстро распутывается.

Название типа – семантическое, его версия – структурная. Структурное версионирование сохраняет идентичность существующих фактов по мере развития приложения. Оно позволяет более новым программам загружать данные, написанные более старыми версиями, но защищает старое программное обеспечение от повреждений из-за неправильной интерпретации новых данных. Версионирование является гораздо более сложной проблемой, когда база данных знает схему каждого типа фактов. Только при использовании хранилища данных, не зависящего от приложений, можно полностью решить проблему версионирования.

# Глава 11

# Коммуникация

Многие архитектурные решения, которые мы принимаем, ограничивают способ доставки сообщений и способ их обработки. Двумя распространенными примерами доставки сообщений являются REST API и сервисные шины. Выбор одного из них часто диктует способ обработки сообщений. В REST API-сообщение отправляется в виде синхронного HTTP-запроса. Сервер обрабатывает запрос и отправляет ответ. В сервисной шине сообщение отправляется асинхронно, путем помещения его в очередь или публикации в теме. Получатель обрабатывает сообщение по мере того, как извлекает его из очереди, и публикует результат, если таковой имеется, для последующего использования. Модели коммуникации и обработки тесно связаны между собой.

Неизменяемые архитектуры дают нам возможность разделить эти две концепции. Мы можем делать выбор в отношении коммуникации на основе того, насколько близко отправитель находится к получателю, контролируются ли они одной и той же организацией, имеют ли они более или менее постоянную связь. С другой стороны, мы можем сделать выбор в отношении обработки информации исходя из того, как предполагается развивать разговор и ожидает ли инициатор ответа. Мы можем выбирать, как обмениваться фактами между узлами, независимо от способа обработки сообщений этими узлами.

Модели, которые мы проанализировали и построили, определяют то, о чем говорят узлы. Они не определяют, как эти узлы будут разговаривать. Чтобы максимизировать автономность, каждый узел будет иметь только то подмножество модели, которое ему необходимо для обслуживания своих пользователей и принятия своих решений. Все они участвуют в обмене информацией для обмена подмножествами друг с другом. Чтобы сделать наиболее подходящий выбор коммуникации, мы должны понимать эти подмножества, потребности каждого узла и ограничения различных коммуникационных протоколов.

## Гарантии доставки

Как учит нас притча о двух генералах, узел не может знать, будет ли получено сообщение, которое он только что отправил. У него нет гарантии, основанной только на отправке сообщения. Вместо этого он узнает об успешной доставке только тогда, когда получает последующее сообщение от удаленного узла.

Знание происходит с задержкой. Гарантии могут быть выполнены только путем повторных попыток до тех пор, пока это знание не будет получено.

К счастью, мы можем построить более надежные гарантии доставки поверх менее надежных протоколов. Это видно из модели OSI[1] для сетей, показанной на рис. 11.1, которая подразделяет стек на семь уровней. Модель описывает множество факторов качества обслуживания, а не только доставку. Чтобы сосредоточиться лишь на гарантиях доставки, нам нужно рассмотреть три уровня: сетевой, транспортный и прикладной.


**Рис. 11.1.** Модель сетевого взаимодействия OSI делит протоколы на семь уровней абстрагирования

На сетевом уровне ни один коммуникационный протокол не дает гарантии доставки. Сеть занимается адресацией, маршрутизацией и коррекцией ошибок, но не доставкой. Она не отвечает за установление долговременных соединений между узлами, повторную попытку передачи неудачных пакетов или даже сообщение об успехе или неудаче.

Следующий, транспортный уровень берет на себя ответственность за сообщение об успешной доставке обратно отправителю. Он обеспечивает подтверждение, но не обязательно опровержение. Часто невозможно доказать,

[1] Сетевая модель OSI (The Open Systems Interconnection model) – сетевая модель стека (магазина) сетевых протоколов OSI/ISO. Посредством данной модели различные сетевые устройства могут взаимодействовать друг с другом. Модель определяет различные уровни взаимодействия систем. Каждый уровень выполняет определенные функции при таком взаимодействии. https://ru.wikipedia.org/wiki/Сетевая_модель_OSI. – *Прим. перев.*

что сообщение *не* было получено. Но в конечном итоге транспортный уровень должен в какой-то момент сдаться. Он не может гарантировать перед отправкой сообщения, что будет продолжать попытки до тех пор, пока сообщение не будет получено.

Только на прикладном уровне протоколы начинают предоставлять такие гарантии. Если сообщение передается надежному протоколу, то он сделает все возможное, чтобы убедиться, что сообщение дошло до адресата. Он будет продолжать пытаться отправить сообщение, пока не убедится, что оно было получено. Он возобновит работу после сбоя питания. Некоторые протоколы даже дают дополнительные обещания относительно порядка, уникальности и задержки доставки. Чем больше обещает надежный протокол, тем дороже он будет стоить. Поэтому мы примем самое слабое обещание, которое можем принять.

## Лучшие усилия

Термин «лучшие усилия» (best effort) – неудачное название. Хотя кажется, что он подразумевает, что не существует больших усилий, которые можно приложить для решения проблемы доставки, на самом деле он означает обратное. Сервис best effort не будет пытаться повторно отправить сообщение после неудачи. Фактически он даже не будет сообщать об успехе или неудаче доставки. Это эквивалент качества обслуживания (quality-of-service – QOS) – пожатие плечами.

Все протоколы в какой-то момент строятся на уровне best-effort. В большинстве современных приложений это обычно означает интернет-протокол (Internet Protocol – IP). Некоторые протоколы расширяют это ограниченное качество обслуживания до прикладного уровня. К ним относятся User Datagram Protocol (UDP) и многоадресная IP-рассылка. Если задержка более важна, чем доставка, эти протоколы являются подходящими вариантами. Они могут использоваться наряду с более надежными протоколами для предоставления таких услуг, как присутствие, потоковая передача и мониторинг состояния.

Чтобы построить протокол лучший, чем best-effort, получатель должен обеспечить обратную связь при получении. Это дает отправителю подтверждение того, что сообщение было доставлено.

## Подтверждение

На транспортном уровне модели OSI некоторые протоколы полагаются на подтверждение того, что пакет получен. Это часто делается для сдерживания обмена данными, задерживая некоторые пакеты до тех пор, пока не будут подтверждены более ранние пакеты. Но во многих случаях это используется еще для установления дуплексного соединения между двумя узлами. Наиболее ярким примером является протокол управления передачей (Transmission Control Protocol – TCP), который построен на базе IP.

Когда установлено дуплексное, или двустороннее, соединение, каждый узел знает, что он может успешно направлять пакеты другому узлу. Это соединение представляет собой туннель, через который можно отправлять и получать сообщения. Узлы могут полагаться на то, что если байты получены, они придут в порядке и с очень низкой вероятностью ошибки. Пока соединение остается открытым и не прервалось, протокол TCP будет повторять попытки передачи

пакетов до тех пор, пока они не будут подтверждены. Вмешательство приложения не требуется.

Многие прикладные протоколы полагаются на дуплексные соединения для обеспечения подтверждения доставки своим потребителям. Примеры слишком многочисленны, чтобы их рассматривать, но наиболее известным и широко используемым является протокол передачи гипертекста (Hypertext Transfer Protocol – HTTP). Несмотря на «гипертекст» в названии, этот протокол стал дефакто стандартом всех видов обмена информацией в Сети, не только HTML, но и SOAP, JSON и gRPC. HTTP поддерживает гарантии доставки, ограничивая изменения состояния узлов при получении различных сообщений.

### Безопасные методы

Спецификация HTTP говорит о двух свойствах методов – безопасности и идемпотентности. Первая категория методов, которую мы рассмотрим, – это методы, обладающие свойством безопасности. *Безопасный метод* не изменяет состояние сервера при получении. Такие методы, как GET и OPTIONS, являются безопасными.

Получив безопасный запрос, сервер может получить информацию, но он не может изменить свое состояние каким-либо видимым образом. Кеширование ответа хотя технически и является изменением состояния, но не является непосредственно заметным для клиента. Поэтому кеширование разрешено для серверов, использующих безопасные методы.

Как клиент вы можете быть уверены, отправляя безопасный запрос, что не вызовете никаких нежелательных изменений состояния. Вы можете повторить GET на другом соединении, если не получили ответа. Теоретически сервер должен ответить таким же образом, предполагая, что за прошедшее время не произошло никаких изменений состояния. Конечно, нет никакого способа для клиента или протокола *обеспечить соблюдение* этого соглашения. Это полностью зависит от сервера – воздерживаться от изменения состояния в ответ на безопасный метод.

### Идемпотентные методы

Второе свойство методов, которое определяет HTTP, – это идемпотентность. Это обещает, что состояние сервера изменится только при первом получении отдельного сообщения, а не при последующем получении. Все безопасные методы по умолчанию являются идемпотентными, сервер не изменит состояние даже при первом получении, не говоря уже о втором. Поэтому вторая категория, которую мы рассматриваем, является идемпотентной, но не безопасной.

Как мы узнали в главе 4, идемпотентность – важное свойство обработчика сообщений. Оно позволяет повторять сообщения, не опасаясь изменения состояния. Если первое сообщение действительно было получено, то второе получение не изменит состояние. `HTTP`, `PUT`, `PATCH` и `DELETE` являются примерами идемпотентных методов.

Хотя идемпотентность необходима для надежной доставки сообщений, причина, по которой эти методы называются идемпотентными, заключается в запрете повторных попыток и восстановления после дублирования. Это прос-

то основано на семантике обновления, исправления или удаления ресурса. Обновление устанавливает ресурс в известное состояние. Исправление устанавливает только некоторые свойства, но все равно для известных состояний. Логически можно предположить, что при дублировании ресурс уже находится в желаемом состоянии. Обновление или исправление ресурса не будет иметь никакого эффекта. Аналогичный аргумент применим и к удалению уже удаленного ресурса.

К сожалению, эта линия рассуждений учитывает только дубликаты без промежуточных изменений. Если ресурс изменился между оригинальным обновлением и дубликатом, то дубликат вернет ресурс в предыдущее состояние. В конечном итоге последовательная обработка сообщения будет игнорировать дубликат, а не применять его.

Рассмотрим пример на рис. 11.2. Первый клиент издает запрос PUT для обновления значения ресурса в «Боб». Эта команда вступает в силу, но соединение прерывается до получения ответа. Тем временем второй клиент отправляет PUT запрос на обновление значения того же ресурса до «Роберт». Этот клиент видит свой ответ. Первый клиент устанавливает новое соединение и повторяет попытку отправки сообщения. Если сервер изменит значение обратно на «Боб», как того требует протокол HTTP, то он не игнорирует дублирующее сообщение.

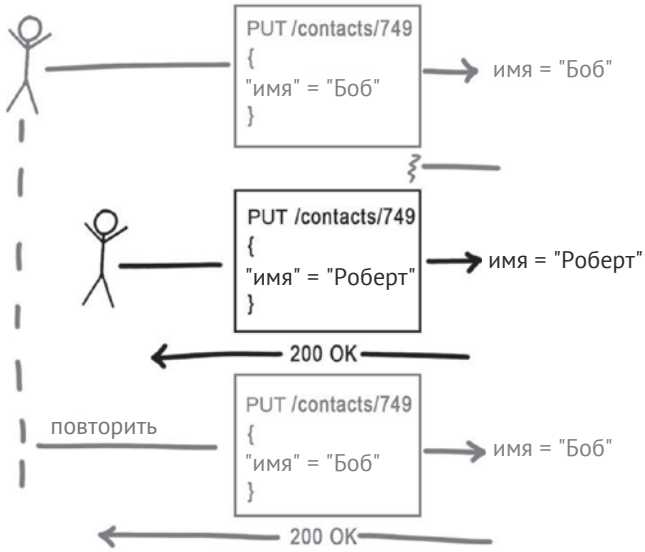


**Рис. 11.2.** Сервер отвечает на дублирующий запрос после того, как вмешался другой запрос

HTTP-сервер ведет себя идемпотентно. И все же система не является в конечном итоге последовательной. Проблема в том, что идемпотентности недостаточно. Как мы показали в главе 4, протокол должен быть коммутативным, чтобы обеспечить конечную согласованность. Если бы сервер на рис. 11.2 отвечал одинаково, независимо от порядка сообщений, то результат был бы лучше. Предположим, что он обработал сообщение «Боб», за которым следует «Роберт», точно так же, как и «Роберт», за которым следует «Боб», например позволяя ресурсу находиться в состоянии суперпозиции двух значений-канди-

датов. Тогда последующее получение дублирующего сообщения будет просто проигнорировано. Вспомните диаграмму из главы 4, повторенную на рис. 11.3, иллюстрирующую это решение.


**Рис. 11.3.** Структура данных, которая позволяет ресурсу иметь несколько одновременных версий, является идемпотентной и коммутативной

Гарантия идемпотентности в HTTP является лишь семантической. Виды действий, которые представляют идемпотентные команды, как правило, приводят ресурс в известное состояние, даже после второго применения. Но они не обеспечивают защиту от дублирования сообщений. HTTP не дает гарантий коммутативности.

### Неидемпотентные методы

Третья категория, которую следует рассмотреть, – это методы, которые не являются ни безопасными, ни идемпотентными. Эти методы не дают никаких гарантий. Они могут менять состояние при каждом получении. Метод `POST` является примером такого типа.

Семантика POST делает вероятным, что изменение действительно произойдет. В ответ на этот запрос сервер создает новый ресурс и возвращает его идентификатор в ответе «201 Created». Предположительно идентификатор ресурса не был известен клиенту до запроса. Если бы он был известен, то в ответе не было бы необходимости указывать URL. Мы можем обоснованно предположить, что в большинстве реализаций POST идентификатор был сгенерирован на сервере.

Если идентификатор генерируется на сервере, то клиент мало что может сделать, чтобы предотвратить дублирование. Если соединение будет потеряно до получения ответа «201 Created», клиент не может ничего сделать, кроме как попробовать еще раз или сообщить об ошибке пользователю (который, скорее всего, повторит попытку). Вторая попытка, скорее всего, при-

ведет к созданию второго ресурса. HTTP не дает никаких гарантий, что сервер поступит иначе.

### Повторная попытка в пределах соединения

Какой бы ни была гарантия изменения состояния метода HTTP, транспортный уровень может только обеспечить подтверждение в рамках соединения. Соединения относительно недолговечны и находятся полностью в памяти. Они представляют собой однопоточный коммуникационный канал связи между двумя партнерами.

Приложениям, построенным на протоколах, ориентированных на соединения, не нужно повторять сообщения в пределах одного и того же соединения. TCP гарантирует порядок передачи байтов, что подразумевает, что повторное сообщение не будет получено сразу после оригинального сообщения. Но если соединение обрывается, то все ставки сделаны. Отправитель не знает, были получены неподтвержденные сообщения или нет.

Соединение может использовать подтверждение доставки, чтобы гарантировать, что сообщение *было* получено. Оно не может гарантировать, что сообщение *будет* получено в какой-то момент в будущем. Подтверждение является необходимым, но не достаточным условием для долговечности. В то время как HTTP только передает семантику соединения с транспортного уровня, другие протоколы прикладного уровня добавляют долговечность.

## Долговечные протоколы

Когда пользователь приложения инициирует команду, он хотел бы иметь определенную уверенность в том, что эффект от команды будет длительным. Протоколы, описанные в предыдущем разделе, просто заставляют их ждать, пока удаленный партнер подтвердит получение сообщения. Но это лишает пользователя некоторой автономии. Он больше не может принять решение и отдать команду, не привлекая удаленного партнера. Чтобы иметь максимальную автономию, он должен иметь возможность работать в изоляции. Поэтому он требует большего от своего протокола.

*Долговечный* протокол – это протокол, который гарантирует, что сообщение в конечном итоге будет доставлено. Подтверждение доставки необходимо, но не достаточно. Долговечные протоколы должны продолжать повторные попытки до получения такого подтверждения, даже в течение длительных периодов времени или при перебоях в подаче электроэнергии. Поэтому долговечные протоколы требуют долговечного хранения на стороне отправителя, которое может быть обеспечено только на прикладном уровне модели OSI. Две наиболее распространенные формы долговременного хранения сообщений – это очереди и темы. Популярными примерами этих форм являются расширенный протокол очереди сообщений (Advanced Message Queueing Protocol – AMQP) и Apache Kafka.

### Очереди

AMQP – это стандартный протокол прикладного уровня для обмена сообщениями в очередях. Он реализован в таких системах очередей, как RabbitMQ,

Apache ActiveMQ и Azure Service Bus. AMQP – это настраиваемый протокол, предлагающий несколько уровней обслуживания. Некоторые из этих уровней обслуживания обеспечивают *по крайней мере однократную доставку* или гарантию того, что отправитель будет продолжать попытки до тех пор, пока сообщение не будет получено. Это обещание сохраняется за пределами одного соединения или сессии. Оно выживает даже после отключения электричества.

Чтобы обеспечить эту гарантию, реализации AMQP хранят сообщения на стороне клиента. Когда пользователь публикует сообщение провайдеру AMQP, он сохраняет сообщение немедленно, до обращения к удаленным партнерам. В этот момент приложение может быть уверенным в том, что сообщение будет доставлено. Движок начинает фоновый процесс создания соединения, передачи сообщений и получения подтверждений. Как только сообщение подтверждено, оно может быть удалено из клиентского хранилища. До этого момента оно должно сохраняться.

### Темы

Если AMQP определяет очереди, то Kafka определяет темы. Тема – это постоянный поток записей. В отличие от очереди, записи в теме не удаляются, когда они потребляются. Это позволяет теме Kafka поддерживать несколько подписчиков, каждый из которых получает все сообщения.

В дополнение к нескольким подписчикам сохранение сообщений позволяет темам обеспечивать более надежную гарантию доставки. Поскольку все прошлые сообщения все еще хранятся в памяти, тема может определить, является ли сообщение дубликатом. Она может игнорировать дубликат, предотвращая его от отправки подписчикам. Этот уровень гарантии называется *однократной доставкой*.

Обнаружение дубликатов длится только до тех пор, пока сообщения сохраняются. Не все реализации тем хранят сообщения неограниченно долго. Темы Kafka, например, имеют настраиваемый период хранения, который по умолчанию составляет 7 дней. Если дублирующее сообщение приходит после истечения периода хранения, то оно будет отправлено подписчикам.

Для неизменяемых архитектур достаточно хотя бы однократной доставки. Такие приложения основаны на структурах данных, которые сохраняют исторические факты неограниченное время. Если дубликат получен, он будет обнаружен, поскольку факт уже находится в хранилище. А так как записи идентифицируются по адресу, основанному на их содержании, коллизии предотвращаются на уровне хранения. Даже если долговечный протокол может предложить гарантии доставки ровно один раз, включение такой конфигурации может оказаться столь же дорогим, сколь и ненужным.

## Обработка сообщений

В дополнение к гарантиям доставки при оценке коммуникационных протоколов мы также должны учитывать время обработки сообщений. *Синхронные* протоколы требуют, чтобы сообщение обрабатывалось сразу после получения, так как пир (одноранговый узел) активно ожидает результат. *Асинхронные* протоколы позволяют получателю обработать сообщение позже. Эти протоколы,

как правило, предлагают большую автономность, поскольку удаленные узлы не ждут друг друга.

Приложения, основанные на неизменяемой архитектуре, склонны ценить автономность больше, чем большинство других факторов. Каждый узел имеет именно то подмножество информации, которое ему необходимо для поддержки решений, принимаемых им и его пользователями. Поэтому асинхронные протоколы имеют тенденцию быть предпочтительными.

## Большинство протоколов являются асинхронными

Выбор между синхронной и асинхронной обработками сообщений не является полностью изолированным от выбора гарантии доставки. Протокол, предлагающий только доставку best-effort, не будет информировать клиентский узел об успехе или неудаче сообщения. Он, конечно, не будет ждать, пока сервер обработает запрос. Такие протоколы поддерживают лишь асинхронную обработку сообщений.

С другой стороны, протоколы, предлагающие гарантии долговечности, обещают это сразу после сохранения сообщения на стороне клиента. Фактическая связь с сервером может произойти вскоре после этого или может быть отложена на длительный период времени. У протокола нет способа сигнализировать приложению клиента о том, что сообщение было доставлено. Поэтому такие протоколы обычно не требуют, чтобы удаленный партнер немедленно обработал запрос, и, как правило, являются асинхронными по своей природе. Неизменяемые архитектуры предпочитают подобные протоколы.

Только в центре, где клиентское приложение получает подтверждение доставки, имеет смысл требовать синхронной обработки сообщений. Клиентское приложение активно ожидает ответа. Этот ответ вполне может включать результаты обработки сообщения.

## HTTP обычно является синхронным

HTTP по умолчанию является синхронным протоколом. Когда клиент посылает запрос, он ждет ответа от сервера. Этот ответ является как подтверждением доставки, так и результатами обработки сообщения. Коды ответов HTTP включают такую информацию, как был ли создан ресурс (201 Created – создан), был ли клиент авторизован для доступа к этому ресурсу (403 Forbidden – запрещено) или привела ли обработка к конфликту (409 Conflict – конфликт). Эти ответы подразумевают, по крайней мере, некоторую степень синхронной обработки.

HTTP, однако, не требует, чтобы сервер обрабатывал ответ немедленно. Некоторые коды ответа HTTP (в частности, 202 Accepted – принято) предназначены для отражения того, что обработка будет происходить асинхронно. В этом случае информация о результатах процесса не включается в ответ. Он служит лишь подтверждением доставки.

В современных приложениях большая часть трафика через публичный интернет основана на протоколе HTTP. Асинхронные протоколы не так популярны за пределами брандмауэра. AMQP может быть туннелирован через TLS и иногда используется в интернете. Но чаще он хранится в защищенном виде в центре

обработки данных организации или открыт между организациями. Мобильные приложения предпочитают HTTP другим протоколам, а клиенты на основе браузеров используют почти исключительно HTTP. Возможно, в будущем использование асинхронных протоколов в интернете станет более распространенным явлением. Но пока присоединение публичного клиента к серверу обычно происходит по протоколу HTTP. Однако это не означает, что мы должны использовать его синхронно. Даже протоколы запрос/ответ можно использовать асинхронно.

## Синхронизация данных

Слово «синхронизация» – еще один неудачный термин в применении к данным. Синхронизация буквально означает заставить две системы работать в одно и то же время или, по крайней мере, с одинаковой скоростью. Два человека могут синхронизировать свои часы, чтобы они показывали одинаковое время. Но синхронизация данных – это не про время. Цель – автономность, а не синхронность. То, к чему мы стремимся при синхронизации данных, – это согласованность. Если вы зададите двум узлам один и тот же вопрос, они дадут один и тот же ответ. Они могут сделать это потому, что имеют одну и ту же информацию, а не потому, что работают в одно и то же время.

Узлы в неизменяемой архитектуре имеют в своем распоряжении подмножество данных. Это позволяет пользователям и процессам на данном узле принимать решения, не обращаясь к другим узлам. Процедура, которую мы называем синхронизацией данных, – это просто процесс обмена неизменными фактами с одноранговыми узлами, чтобы их структуры данных сходились. Каждый узел будет иметь только то подмножество, которое ему необходимо, но там, где эти подмножества пересекаются, правила бесконфликтной репликации типов данных (CRDT) гарантируют, что согласованность будет достигнута, когда процедура будет завершена.

Опираясь на неизменяемые структуры данных, мы теперь можем самостоятельно решать, какие протоколы использовать в этой процедуре. Какие гарантии доставки нам требуются? Должны ли сообщения обрабатываться синхронно или асинхронно? Ограничены ли мы общими открытыми протоколами или можем выбрать индивидуальные варианты с более желаемыми характеристиками? Будут ли узлы адресуемыми, или нам придется ждать, пока они сами к нам обратятся? Будут ли узлы постоянно подключены, или они будут подключаться только время от времени?

Чтобы ответить на эти вопросы, мы рассмотрим три основных варианта использования. Каждый из них представляет собой различные коммуникационные структуры, которые обычно встречаются в неизменяемых архитектурах. Каждый из них требует несколько иного набора протоколов.

Для связи между серверами внутри организации мы отдадим предпочтение менее вездесущим, но более асинхронным протоколам, таким как AMQP и Kafka. Это поможет нам построить неизменяемую архитектуру микросервисов. Для коммуникаций между организациями мы будем отдавать предпочтение более распространенным технологиям API REST и webhook[1] что

[1] Термин webhook (веб-хук) не имеет устоявшегося перевода на русский язык. Это

приведет к меньшей связанности инфраструктуры. А для периодически подключаемых клиентов, таких как мобильные приложения, и прогрессивных веб-приложений (progressive web apps – PWA) мы будем использовать HTTP в качестве асинхронного протокола.

## В рамках организации

Синхронизация данных внутри организации – это своего рода роскошь. Одна группа контролирует все серверы, все хранилища данных и всю сеть. У нас есть возможность выбрать AMQP или Kafka, если захотим. У нас также есть роскошь быстрого, всегда доступного соединения между микросервисами. Мы не будем злоупотреблять этой роскошью, обращаясь от одного к другому при каждом запросе, но мы можем поддерживать их хранилища данных синхронизированными.

Имея такую роскошь, легко стать самодовольным. Внутриорганизационные архитектуры иногда разделяют базы данных между сервисами. Они часто ослабляют контроль безопасности в пределах межсетевого экрана. И они будут игнорировать вопросы версионности, поскольку могут одновременно развернуть обе стороны соединения. Каждый из этих компромиссов архитектурной целостности достигается ценой будущей гибкости. Они увеличивают связь между сервисами ради удобства. При развертывании микросервисов в рамках одной организации вы можете воспользоваться всеми преимуществами, которые у вас есть, при этом избегая ненужного сцепления.

Чтобы понять, *как* именно мы собираемся синхронизировать данные между микросервисами, мы должны сначала определить, *что* они собой представляют. Затем мы можем проанализировать границы между ними, чтобы выбрать наилучший способ интеграции. Результаты анализа, проведенного в главе 5, являются вашим ориентиром в определении того, где должны быть проведены границы.

### Опорные точки

При создании исторической модели мы определили регионы. Это были области модели, в которых все факты происходят от определенного субъекта (действующего лица). Когда отношения предшественник/преемник пересекают границу между двумя регионами, два субъекта сотрудничают друг с другом. Я называю такие отношения предшественник/преемник *опорными точками*.

Диаграмма, которую мы использовали для представления регионов, воспроизведена на рис. 11.4. Я выделил опорные точки.

технология передачи запрашивающей системе актуальной информации по мере ее появления, своего рода информационный обратный вызов. Далее об этом будет рассказано подробнее. – *Прим. ред.*

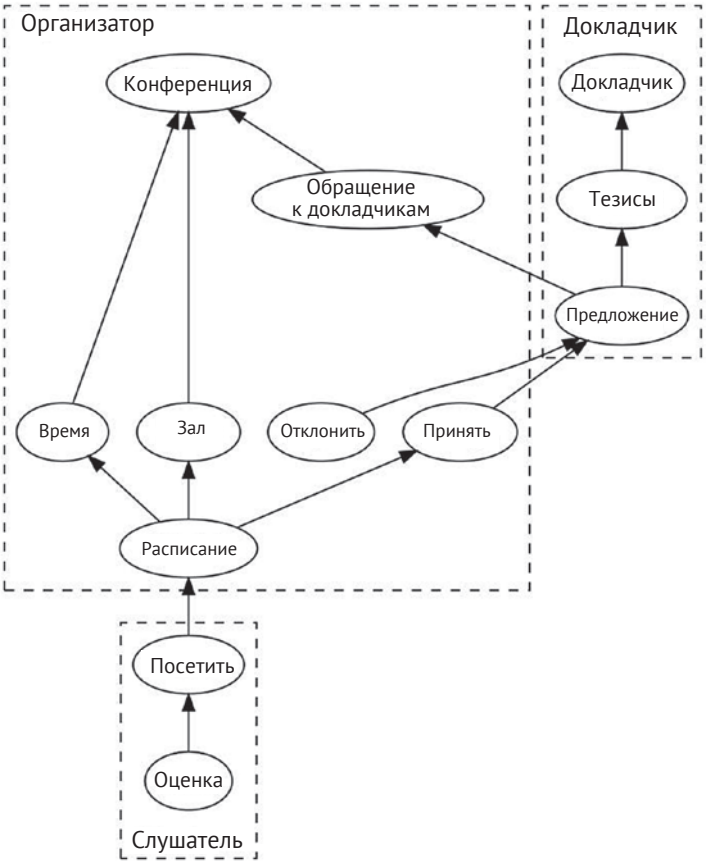


**Рис. 11.4.** Модель, выделяющая опорные точки, где стрелки пересекают границы регионов

На этапе анализа регион представлял собой обязанности одного действующего лица или набора действующих лиц. Когда мы переходим к реализации, то создаем микросервис для каждой группы пользователей. Таким образом, каждый регион теперь представляет собой микросервис. В этом примере организаторы конференции имеют микросервис для сбора предложений и определения расписания. У докладчиков есть микросервис для просмотра приглашений, подачи предложений и получения информации о принятии заявок. Наконец, у посетителей есть микросервис для просмотра расписания конференции, выбора сессий для посещения и выставления оценок. Опорные точки – это точки интеграции между этими микросервисами.

Микросервис в начале опорной точки должен опубликовать факт-предшественник, чтобы микросервис в конце мог подписаться на него. Давайте начнем с самого верхнего предшественника в причинно-следственной цепочке, *приглашения к выступлению*. Этот факт находится в микросервисе *организатора*.

### Многие подписчики

Опорные точки на вершине причинно-следственной цепи, как правило, являются местами, где факты публикуются для нескольких подписчиков. Издатель может не иметь в виду один конкретный случай использования, и будущие подписчики могут быть добавлены в любое время. Но даже если известен конкретный случай использования, как в этом примере, отправка сообщения конкретному подписчику вносит ненужную связь. Поэтому опорные точки верхнего уровня являются хорошими кандидатами на темы, которые предоставляет Kafka.

Микросервис в начале стрелки публикует сообщение в тему, когда создается факт-предшественник. Это сообщение включает в себя всю информацию, содержащуюся в факте и всех его предшественниках. Чтобы вычислить набор всех фактов, включенных в сообщение, выполните *транзитивное замыкание* над предшественниками. Рекурсивно посетите отношения предшественников, пока не будет собран весь набор. Сообщение должно содержать всю эту информацию, и *только* ее.

Две проблемы возникают при публикации сообщения, которое содержит больше, чем транзитивное замыкание предшественников. Первая заключается в том, что сообщение не является детерминированным. Если сообщение содержит внутренние идентификаторы базы данных, время создания или любую другую деталь, которая не является частью фактов, то повторный запуск процесса приведет к другому сообщению. Процесс может быть повторен по любой причине – произошел сбой в инфраструктуре, факт был произведен двумя дублирующимися экземплярами, пользователь нажал на кнопку отправки дважды и т. д. Если возникнет любая из этих ситуаций, мы хотим, чтобы процесс создания сообщения из факта был детерминированным, чтобы последующий потребитель мог применять идемпотентность и игнорировать дубликаты.

Вторая проблема заключается в том, что сообщение может содержать информацию в фактах-преемниках. Преемники могут быть созданы либо до, либо после публикации сообщения. Если сообщение содержит информацию о преемниках, то подписчики не узнают о новых преемниках, созданных после факта. В примере на рис. 11.4 транзитивное замыкание факта *созыва докладчиков* включает *конференцию*. Оно не включает *время* или *помещения*.

Если докладчикам необходимо знать (по каким-то причинам) количество комнат на конференции, эта информация может быть доступна или недоступна на момент публикации. Если комната будет добавлена позже, они об этом не узнают.

Есть некоторые преемники, о которых вы захотите, чтобы подписчики узнали. Например, *дата конференции* (не показана на рис. 11.4) будет важной, чтобы понять, стоит ли подавать заявку. Учитывая, что это, скорее всего, будет изменяемым свойством конференции, она будет смоделирована как преемник. У аналитиков есть два варианта решения этой проблемы – они могут либо превратить преемника в *предшественника* опубликованного факта, либо опубликовать преемника.

Чтобы превратить преемника в предшественника, примените шаблон *Transaction*, описанный в главе 8. Пример показан на рис. 11.5. Опубликованный факт представляет собой транзакцию, которая объединяет всех преемников, актуальных на момент публикации. Это включает этих преемников в транзитивное замыкание. Организатор может изменить дату или место проведения конференции после публикации объявления о наборе докладчиков, но теперь у него есть четкая информация, которой должен располагать докладчик, когда предлагает свою презентацию. Он может использовать эту информацию, чтобы опубликовать новый вызов для докладчиков и связаться с докладчиками, которые ответили на предыдущий вызов.

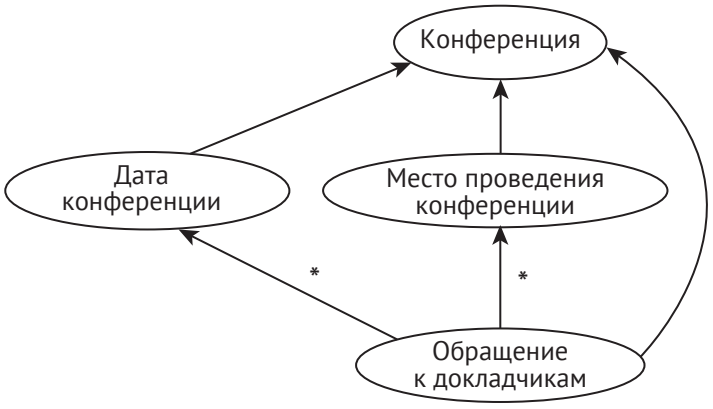


**Рис. 11.5.** Вызов докладчиков – это транзакция, которая фиксирует дату и место проведения конференции

Чтобы опубликовать преемника, создайте дополнительную тему. Подписчики опорной точки могут также подписаться на эту тему. Они будут сопоставлять сообщения между двумя темами, используя их общего предшественника, в данном случае *конференцию*. Если дата конференции изменится после публикации объявления о наборе докладчиков, то подписчик увидит это изменение и обновит свое хранилище данных.

## Ответы

Факты на хвосте опорной точки представляют собой ответы на сообщения. Ответы направляются предыдущим микросервисам. Поэтому имеет смысл использовать очереди вместо тем для этих типов сообщений. Производитель исходного сообщения включает имя очереди ответов. Подписчики отправляют ответные сообщения в данную очередь. Это позволяет управлять связью между издателем и подписчиком, поскольку имя очереди предоставляется динамически. На рис. 11.4 факт *предложения* является ответом на обращение к докладчикам. Он направлен в микросервис *организатора*. Этот микросервис будет создавать очередь, специально предназначенную для приема предложений, и управлять ею.

Ответное сообщение, как и исходное, составляется из транзитивного замыкания факта. В данном случае это означает, что *предложение* содержит инфор-

мацию о *тезисах* и *докладчике*. Если требуются преемники (например, имя докладчика), тогда ответ должен следовать шаблону Transaction, как показано на рис. 11.6. Обратите внимание также на то, что сообщение содержит *обращение к докладчикам*. Исходное сообщение является предшественником и поэтому выступает частью транзитивного замыкания. Это дает исходному микросервису достаточную информацию для соотнесения ответов.

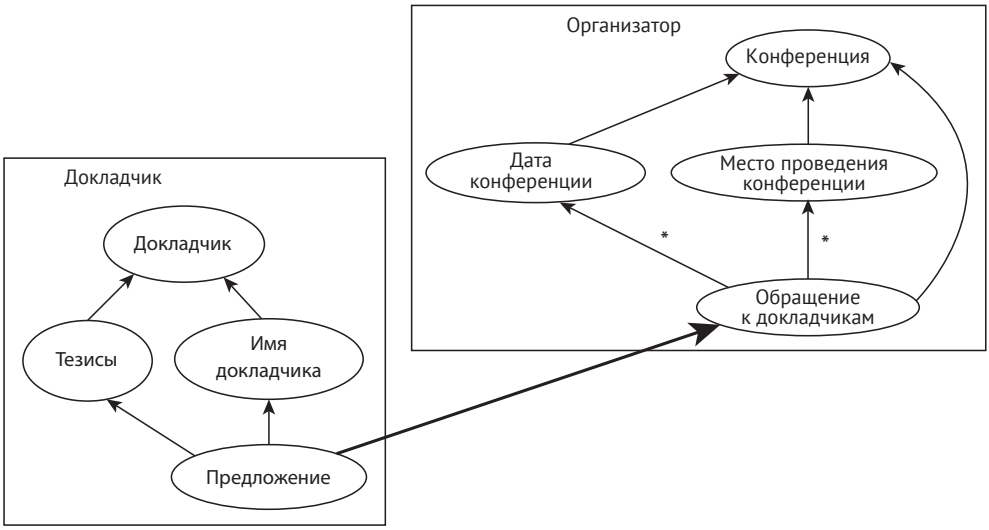


**Рис. 11.6.** В предложении собраны все факты, необходимые организатору для принятия решения

Поскольку микросервис докладчика знает о теме «*Обращение к докладчикам*», очень заманчиво, чтобы он также знал об *очереди предложений*. Кажется, что это не увеличит связь между двумя сервисами. Но это было бы ошибкой. Только микросервис организатора знает, *как* он будет реагировать на предложения. Он может изменить топологию в будущем релизе. Лишь он должен отвечать за принятие решения о местоположении очереди ответов.

Опорные точки, расположенные ниже в причинно-следственной цепи, также представляют собой ответы. На рис. 11.4 факты *отклонения* и *принятия* являются ответами на *предложение*. Эти виды ответов должны следовать одной и той же схеме. Когда микросервис докладчика генерирует сообщение о предложении, он включает имя *своей собственной* очереди ответов.

## Уведомления

Не все ответы являются прямыми преемниками опорных точек. И не все двусторонние разговоры между микросервисами выглядят как стрелки, пересекающие границы регионов в обоих направлениях. Иногда отношения скрыты немного глубже в модели.

Каждый разговор между микросервисами заканчивается сообщением, на которое нет ответа. Эти сообщения служат только в качестве *уведомления*. Они информируют получателя о результате процесса. В модели они выглядят

как листья, расположенные ниже опорных точек. Примером является факт оценки на рис. 11.4.

Когда участник конференции оценивает сессию, на которой он присутствовал, он просто дает обратную связь организатору. Он не ожидает никакого ответа на эту оценку. Поэтому под фактом *оценки* нет опорной точки, указывающей на то, что разговор продолжается. Оценка переносится в очередь, которую предоставляет организатор, как и любой другой ответ. Имя этой очереди будет включено в сообщение *расписания*, так как она является предшественником ближайшей очереди.

## Между организациями

Когда серверы не находятся под нашим непосредственным контролем, мы теряем часть той роскоши, которую могли бы иметь в рамках одной организации. Мы больше не можем выбирать из всех доступных протоколов, чтобы выбрать наиболее подходящий. И у нас нет возможности управлять способом моделирования систем коллег. Партнеры могут даже не использовать неизменяемую архитектуру. Мы адаптируемся, внедряя дополнительные ограничения и трансформируя наши сервисы, чтобы они были более удобными.

Одним из ограничений при пересечении границ организаций является то, что коммуникационные протоколы должны поддерживаться обеими сторонами. Это обычно означает, что асинхронные протоколы, такие как AMQP, заменяются синхронными протоколами, такими как HTTP. Время обработки сообщений не является проблемой, это просто адаптация. Более широко распространенными протоколами сегодня являются те, которые поддерживают синхронную обработку по умолчанию. Компромисс часто может быть достигнут путем использования HTTP в асинхронном режиме.

### Асинхронный через HTTP

Внешним организациям нередко требуется публиковать сообщения для ваших сервисов. Семантически это команды, предписывающие вашему сервису выполнить какую-либо бизнес-функцию. В исторической модели это просто предшественники опорной точки, созданные в удаленном регионе модели. Если бы мы работали исключительно в рамках одной организации, мы могли бы выбрать тему или очередь для публикации этих сообщений. Но поскольку мы предоставляем конечную точку для партнера, то будем использовать HTTP. Мы можем разработать конечную точку с дополнительными ограничениями, не входящими в спецификацию HTTP, чтобы заставить ее работать с неизменяемыми архитектурами.

Согласно спецификации HTTP, `POST` не обязан быть ни безопасным, ни идемпотентным. Однако конечная точка, предоставляемая партнерским организациям, явно выиграет от идемпотентности. Это не исключает HTTP-запросов POST. Это лишь означает, что мы реализуем сервер, чтобы поддерживать более сильные гарантии, чем требует спецификация.

Во-первых, мы гарантируем, что тело сообщения содержит достаточно информации для генерирования уникального идентификатора. Когда мы получим запрос, мы создадим факт. Содержание этого нового факта должно быть полностью определено содержанием сообщения. Мы не будем использовать

время получения, идентификатор, сгенерированный сервером, или любые другие недетерминированные данные для создания этого факта. Это гарантирует, что если партнер повторит данный запрос, он получит тот же факт. Это первый шаг к тому, чтобы сделать POST идемпотентным.

Во-вторых, мы генерируем URL ресурса, используя только информацию из нового факта, и вычисляем транзитивное замыкание нового факта, чтобы найти граф всех предшественников. Извлекаем поля из этих предшественников и собираем их в путь. Добавьте этот путь к имени хоста открытой конечной точки, чтобы получить URL. Предполагая, что мы использовали все поля, мы создаем соответствие один к одному между фактами и URL. Когда партнер сделает последующий вызов конечной точки, мы сможем извлечь компоненты из пути и восстановить факт.

И в-третьих, отвечайте на `POST` сразу после сохранения нового факта. Не ждите, пока запрос будет обработан. Перед сохранением факта у вас будет возможность запустить правила авторизации, чтобы убедиться, что партнер уполномочен сделать этот запрос. Но нет необходимости ждать, пока запрос будет обработан. Вы можете завершить обработку асинхронно.

Конечная точка, реализованная в соответствии с этими ограничениями, будет идемпотентной. Любой последующий `POST` того же запроса будет выдавать существующий факт. Поскольку сервис использует хранилище с адресом содержимого, он распознает, что факт уже существует. Он просто ответит тем же URL, который был получен первоначально.

Такая конечная точка также является долговечной. Она не отвечает до тех пор, пока факт не будет сохранен. Побочным эффектом хранения факта может быть также добавление его в тему или очередь для дальнейшей обработки. Подтверждение доставки «201 Created» указывает на то, что это сохранение произошло и было зафиксировано. Отправитель может прекратить отправку в этот момент, сообщение было сохранено.

Наконец, эта конечная точка не зависит от местоположения. URL не содержит никаких генерируемых сервером идентификаторов. Если бы запрос был обработан другим сервером, он выдал бы тот же URL и тот же факт. Мы можем свободно реорганизовать нашу инфраструктуру, переключиться на резервный центр данных или зеркалировать запросы в разные географические регионы. Ни одна из этих деталей реализации не будет видна нашим партнерам.

### Webhook

Если бы наша инфраструктура была полностью под нашим контролем, мы могли бы просто отправлять ответы в очередь. Однако при работе с партнерами мы иногда не можем позволить себе роскошь использовать протоколы очередей. И все же мы хотим передавать имена очередей через организационные границы, чтобы уменьшить связь между одноранговыми сервисами.

Эквивалентом очереди ответов в HTTP является webhook. Это HTTP-конечная точка, предназначенная для использования в качестве обратного вызова (callback), места, куда отправляются ответы. Одна служба регистрирует webhook в другой службе, сообщая ей URL конечной точки. Другая служба отправляет запросы `POST` в эту конечную точку всякий раз, когда появляется новая информация по теме.

Ответ в исторической модели появляется как непосредственный или конечный преемник опорной точки. Мы должны генерировать webhook на основе предшественника опорной точки. Как описано ранее, вычислите транзитивное замыкание предшественника и извлеките все поля этих фактов. Сконструируйте путь и добавьте его к имени хоста. Теперь этот URL-адрес можно использовать в качестве адреса обратного вызова. Сервис, прослушивающий этот хост, может восстановить предшественника из пути.

Поскольку путь содержит всю информацию, необходимую для реконструкции предшественника, тело сообщения не обязательно должно включать его. Тело сообщения – это вся информация, необходимая для создания факта-преемника, за исключением предшественника, указанного в пути. Сервис, обрабатывающий webhook, будет следовать всем ограничениям командной конечной точки, описанным ранее, чтобы гарантировать, что ответы будут идемпотентными, долговечными и независимыми от местоположения.

## Эмуляция REST

Во многих случаях интеграции организация, принявшая неизменяемую архитектуру, будет интегрироваться с той, которая этого не сделала. У нас может не быть возможности определить API таким образом, чтобы он хорошо работал с неизменяемой архитектурой. Возможно, нам придется придерживаться API, который определил партнер, или предоставить более привычный для него API. В таких ситуациях мы можем потреблять и реализовывать REST API из неизменяемых сервисов.

Чтобы использовать REST API из неизменяемой модели, примените шаблон Outbox, как описано в главе 8. Шаблон Outbox создает мост между исторической моделью и сторонним API. Вызывающая сторона сопоставляет факты, о которых необходимо знать партнеру, с вызовами API. Она записывает журнал ответов на эти вызовы API, индексированный хешем фактов. Хотя этот шаблон не может превратить REST API в идемпотентный, долговечный обмен данными, он обеспечивает хотя бы небольшую защиту от сбоев инфраструктуры. Остальное зависит от партнера.

Чтобы создать REST API с неизменяемой моделью, мы применяем структурные модели главы 8, преобразуя все входящие запросы в семантически эквивалентные факты. `POST` привязывается к факту шаблона Entity и, вероятно, к одному или нескольким фактам шаблона изменяемых свойств Mutable Property. `PUT` или `PATCH` отображается на одно или несколько изменяемых свойств. `DELETE` генерирует факт шаблона Delete. Основываясь на семантике области, могут быть задействованы и другие шаблоны.

По возможности генерируйте URL-адреса, как описано ранее, используя только информацию, найденную в переходном замыкании. В идеале вся информация, необходимая для генерации факта, будет присутствовать в запросе. Это позволило бы создать действительно идемпотентный API. Однако это не всегда возможно. В частности, факты с изменяемыми свойствами не могут быть сгенерированы только на основе желаемого значения этих свойств. Они должны знать своих предшественников, которые не всегда традиционно предоставлены в REST API.

Чтобы найти предшественников изменяемого свойства, сервис должен выполнить запрос. Найдите все факты, которые не были заменены.

```
query valuesOfProperty(e: Entity) {
  match p: EntityProperty where p.entity = e
       such that not exists n: EntityProperty where n.prior = p
}
```

Если в результате запроса получен один факт и значение этого факта совпадает с желаемым значением в PUT или PATCH, то игнорируйте запрос. Свойство уже однозначно имеет желаемое значение. Но если количество результатов *не равно* 1 или текущее значение другое, то создайте новый факт свойства, имеющий эти результаты в качестве предшественников. Этот алгоритм позволяет клиенту разрешать конфликты, подставляя желаемое значение. К сожалению, он не фиксирует, каким *на самом деле* клиент *считал* исходное значение, и поэтому не записывает реальный причинно-следственный граф. Только клиент, участвующий в неизменяемой модели и использующий соответствующим образом разработанный API, может сделать это.

Чтобы получить ресурс из исторической модели, вам нужно выполнить несколько запросов свойств. Сгенерируйте факт начальной сущности из URL, как описано ранее. Затем выполните запросы для всех свойств, которые вы собираетесь получить. Если любой из этих запросов возвращает более одного результата, примените функцию разрешения конфликтов, чтобы определить желаемый результат. Потребители REST не привыкли к тому, что свойства имеют более одного значения. *Не* сохраняйте результаты разрешения конфликта. GET должны быть безопасными.

REST API, созданный на основе исторической модели, скомпрометирует некоторые из преимуществ неизменяемости. Он будет настолько же идемпотентным, как и традиционная реализация REST. Он не будет иметь гарантий коммутативности полной неизменяемой архитектуры. Но он будет более привычным для партнеров, которые еще не приняли эти стратегии.

## Клиенты, подключающиеся время от времени

Третьим распространенным сценарием интеграции с неизменяемой архитектурой является поддержка автономных мобильных или веб-клиентов. В то время как большинство мобильных приложений и веб-сайтов, используемых сегодня, должны иметь постоянное соединение с внутренним API, автономные клиенты могут взаимодействовать с пользователем даже при прерывании этого соединения. У них есть собственное хранилище, собственная очередь исходящих сообщений, и они могут участвовать в обнаружении и разрешении конфликтов. Собственные мобильные приложения имеют возможности хранения данных от операционной системы, веб-клиенты могут использовать расширенные возможности браузера, работая как прогрессивные веб-приложения (progressive web apps – PWA).

Мобильные и веб-приложения, разработанные для использования таким образом, обычно *сначала работают в автономном режиме*. Все данные, представляемые пользователю, загружаются из локального хранилища, а не из вызова API. Каждое действие пользователя хранится локально и передается в очередь, а не отправляется на сервер. Синхронизация локального хранилища с историей сервера происходит в фоновом режиме. Пользователь может видеть ход этой деятельности, но не блокируется ею.

Время от времени количество подключенных клиентов будет значительно превышать количество серверов. Они будут подключаться к сети только с помощью загрузки или закладки от пользователя. Они могут использоваться в течение длительного периода времени или могут посетить сеть один раз и быстро покинуть. Когда клиент покинет экосистему, сервер не получит никакого уведомления. Поэтому было бы расточительно для серверов отслеживать метаинформацию от имени клиентов. Протокол для синхронизации случайно подключившегося клиента возлагает бремя хранения информации полностью на клиента.

### Очередь на стороне клиента

Когда пользователь взаимодействует с клиентским приложением, оно генерирует факты. Эти факты будут храниться в локальном поднаборе исторического графа. Они также будут добавлены в исходящую очередь. Пользователю разрешено продолжать взаимодействие с приложением, как только факт будет сохранен, а сообщение поставлено в очередь. Ему не нужно ждать, пока оно будет отправлено на сервер.

Мобильные приложения могут использовать локальную базу данных SQLite, Core Data или Realm для хранения фактов и очередей. Для проектирования хранилища фактов обратитесь к советам, приведенным в главе 10. Исходящая очередь – это просто запись о том, какие факты еще не были отправлены на сервер. Это может быть простая таблица внешних ключей к хранилищу фактов.

Прогрессивные веб-приложения могут использовать IndexedDB для хранения фактов и очередей. Эта функция браузера не так богата, как база данных SQL. Вместо этого она представляет собой просто набор коллекций пар имя/значение. Рассмотрите возможность использования одной коллекции для каждого типа фактов. Ключами этих коллекций являются хеши фактов. Кроме того, PWA имеет коллекцию для исходящей очереди, индексируемую по монотонно возрастающему ключу.

Чтобы отправить исходящие сообщения на сервер, мобильное приложение или PWA вызывает HTTP-конечную точку. Это неполная REST-конечная точка, обеспечивающая обычную семантику `POST`, `PUT`, `PATCH` и `DELETE`. Такой тип конечной точки ставит под угрозу ценность неизменяемой архитектуры и предназначен для использования клиентами, которые не участвуют в исторической модели. Вместо этого это более ограниченная конечная точка, в которую сообщения могут быть отправлены `POST` идемпотентным и коммутативным способом, как описано для передачи команд внутри организации.

Чтобы уменьшить задержку и наиболее эффективно использовать сеть, клиенты будут объединять несколько исходящих фактов в один запрос. Содержимое `POST` будет представлять собой набор фактов различных типов. Мой любимый способ кодирования пакета – это объект JSON, в котором ключи представляют собой хеши, закодированные по стандарту base-64. Это позволяет серверу легко находить входящие факты по их хешу и гарантировать, что факт не будет без необходимости дублироваться в пределах одного и того же пакета. Тело каждого факта содержит тип, поля и хеши его предшественников.

Если предположить, что предшественники уже были известны серверу до начала загрузки, то ему не составит труда найти их по хешу и установить взаи-

мосвязь в своей собственной базе данных. Однако на практике это предположение не может быть гарантировано. Клиент может не общаться с одним и тем же сервером от одной сессии к другой. Серверы могут быть распределены по разным центрам данных для обеспечения резервирования или географической близости. Поэтому целесообразно включать транзитивное замыкание по предшественникам всех исходящих фактов. Вот почему важно устранить ненужное дублирование внутри пакета.

Когда сервер получает пакет, он должен хранить каждый из фактов по очереди. Хранение фактов требует выполнения авторизационных запросов и установки внешних ключей. По этим причинам сервер должен уже хранить предшественников. Поэтому он обрабатывает входящий пакет *в топологическом порядке*. Он рекурсивно посещает всех предшественников перед обработкой каждого сообщения. Когда он посещает факт, то сначала просматривает временную структуру данных, чтобы узнать, не посещал ли он уже этот факт. Если нет, он проверяет хеш, а затем ищет в своей собственной базе данных эту запись. Если она присутствует, он идет дальше. Если нет, он запускает правила авторизации и сохраняет новый факт.

Когда все готово, сервер отвечает сообщением «200 OK». После этого клиент может удалить все факты, отправленные в пакете, из своей очереди исходящих сообщений. Клиент продолжает до тех пор, пока очередь не будет пуста.

### Закладка на стороне клиента

Поскольку клиентов больше, чем серверов, вся метаинформация хранится у клиента. Это включает в себя исходящую очередь, которую мы только что обсуждали. Она также включает информацию о *входящих* фактах. Вместо того чтобы хранить очередь для каждого клиента на сервере, каждый клиент хранит свою собственную *закладку*.

Закладка – это место в последовательности фактов. Она идентифицирует последний факт, который клиент получил и сохранил. Клиент может запросить пакет фактов *больше, чем* данная закладка, и сервер ответит как коллекцией фактов, так и новой закладкой. Эта новая закладка соответствует последнему факту в пакете.

Поскольку нам нужно знать, какие факты идут после данной закладки, эти идентификаторы должны быть *полностью упорядочены*. Полный порядок – это порядок, который позволяет нам сравнивать любые два элемента в последовательности. Мы можем точно сказать, находится ли один элемент до или после другого. Во всех остальных смыслах, однако, факты являются *частично упорядоченными*. Вы знаете, что предшественник появился раньше преемника, но вы не можете сравнить два факта, которые не связаны между собой причинно-следственными связями. Более того, факты обычно идентифицируются по их хешу, который не подчиняется никакому порядку. Поэтому нам нужен новый метод идентификации фактов для использования с закладкой.

Идентификаторы фактов в последовательности должны монотонно возрастать. Более поздним фактам нельзя присваивать идентификаторы, меньшие или равные, чем у более ранних. Если это условие будет нарушено, то клиент, использующий закладку из более раннего факта, будет пропускать более поздние факты при последующих запросах. Одних временных меток для этой цели недостаточно, так как два факта могут быть сохранены в одно и то же время.

Автоинкрементный идентификатор – лучший выбор. Даже в этом случае необходимо принять дополнительные меры предосторожности, чтобы избежать чтения более позднего идентификатора до того, как более ранние идентификаторы были зафиксированы. Одной из таких мер предосторожности является удаление фактов из конца пакета, пока не будет найден один, достаточно старый для того, чтобы параллельные записи успели завершиться. Это подразумевает, что клиенты могут не получить абсолютно последнюю информацию до последующего чтения, но защищает от записей, которые происходят вне порядка присвоения идентификаторов.

Наложение общего порядка на частично упорядоченную коллекцию имеет серьезный недостаток. Это означает, что закладки зависят от местоположения. Если мобильное устройство или PWA подключается к другому серверу при последующей выборке, то закладка, полученная в результате предыдущей выборки, будет бессмысленной. Различные внутренние узлы могли расположить частично упорядоченные факты в разном общем порядке. По этой причине клиенту необходимо хранить отдельную закладку для каждого хранилища данных, к которому он обращается.

Центр обработки данных, имеющий кластер серверов с балансировкой нагрузки, все из которых используют общую базу данных, не является проблемой. Независимо от того, какой сервер использует клиент, общая база данных генерирует монотонно возрастающие идентификаторы. Проблема возникает только тогда, когда серверы используют различные хранилища данных. Таким образом, закладки действительно относятся к каждой базе данных, а не к каждому серверу. Каждая база данных должна генерировать свой собственный уникальный идентификатор и использовать его, чтобы отличать свои закладки от закладок других баз данных.

Клиент отправляет все свои закладки вместе с запросом. Сервер определяет, какая закладка связана с используемой базой данных. Если у клиента нет закладки для этой базы данных, он начинает с самого начала. Затем сервер отвечает пакетом фактов, идентификатором базы данных и новой закладкой. Клиент сохраняет все эти факты и обновляет свою закладку для этой конкретной базы данных. Это повторяется до тех пор, пока запрос не даст новых фактов.

В большинстве сетевых топологий включение идентификатора базы данных – это излишняя предосторожность. Вся совокупность мобильных клиентов может обслуживаться из одной базы данных. До тех пор, пока это остается верным, у каждого клиента будет одна закладка, которая неуклонно движется вперед. Однако если когда-нибудь наступит день, когда база данных должна будет переключиться на резерв из-за отказа, то нет никакой гарантии, что порядок вставок будет согласован между текущей и резервной базами данных. Клиентам придется заново загружать набор данных, но они смогут обнаружить и игнорировать дубликаты. Они также гарантированно не пропустят никакой информации, поскольку данные в разных базах данных хранятся в разном порядке.

## Выбор подмножества

Мобильный или PWA-клиент редко нуждается в получении всего содержимого модели данных. У таких клиентов будет один пользователь, и этот пользователь будет иметь доступ только к подмножеству данных. Изредка подключа-

емые клиенты должны получать только те факты, которые нужны их пользователям. Исходя из понимания модели и того, как она будет использоваться, мы можем разделить факты на подмножества. Конкретный пользователь будет иметь доступ к небольшому количеству этих подмножеств. Поэтому клиенту нужно будет следить за отдельными закладками. У него есть одна закладка на базу данных подмножества.

Подмножество модели может быть определено одним корневым фактом. Подмножество включает всех прямых и косвенных преемников этого корня. Представьте себе конус, простирающийся вниз от корня и собирающий в себя все, к чему он прикасается, как показано на рис. 11.7. Это и есть подмножество модели, которое необходимо пользователю для взаимодействия с этим корнем.

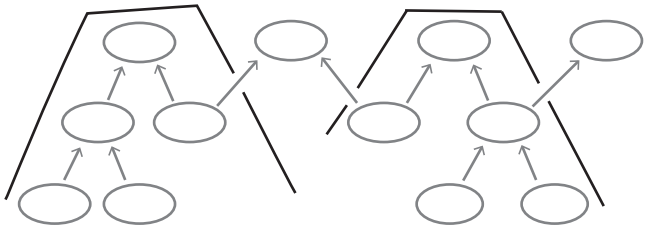

**Рис. 11.7.** Подмножество модели – это конус прямых и косвенных преемников заданного корня

Есть два вида фактов, которые делают хорошие корни подмножеств – группы и периоды. Группа – это факт верхнего уровня, участвующий в шаблоне Membership, определенном в главе 8. Факт членства имеет двух предшественников – группу и пользователя. Он предоставляет пользователю членство в группе и, следовательно, доступ к ее ресурсам. Факты членства часто определяют правила авторизации и распределения. Пользователю не требуется видеть факты за пределами группы, членом которой он является.

Период – это факт уровня, близкого к верхнему, участвующий в шаблоне Period. Этот факт разбивает преемников по времени. Естественные часы в проблемной области движутся вперед и указывают на текущий период. Это может быть дата ведения бизнеса в ресторане или семестр в школе. Новые факты добавляются к текущему периоду. Поэтому клиентские приложения могут фокусироваться только на последних периодах и игнорировать старые.

Когда пользователь взаимодействует с эпизодически подключающимся клиентом, ему нужны только преемники групп, к которым он принадлежит, и самые последние периоды. Но чтобы понять преемника, ему также необходимо транзитивное замыкание его предшественников. По этой причине подмножество, которое фактически загружается на устройство, включает предшественников этих преемников. Конус возвращается вверх по графу, образуя решетчатую структуру, как показано на рис. 11.8.

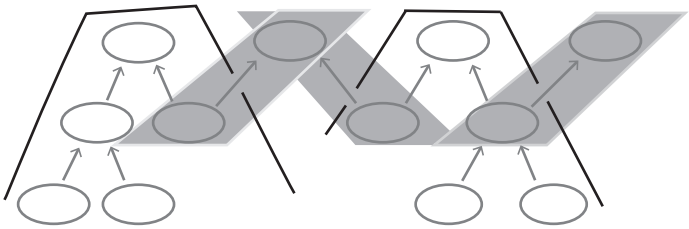

**Рис. 11.8.** Факты, загруженные на устройство, включают в себя конус преемников корня и всех их предшественников

Вспомните пример с секретным каналом, который мы изучали в главе 7. В этом примере создатель секретного канала отправлял приглашение своим коллегам. Затем члены канала могли обмениваться сообщениями друг с другом. Диаграмма показана на рис. 11.9.

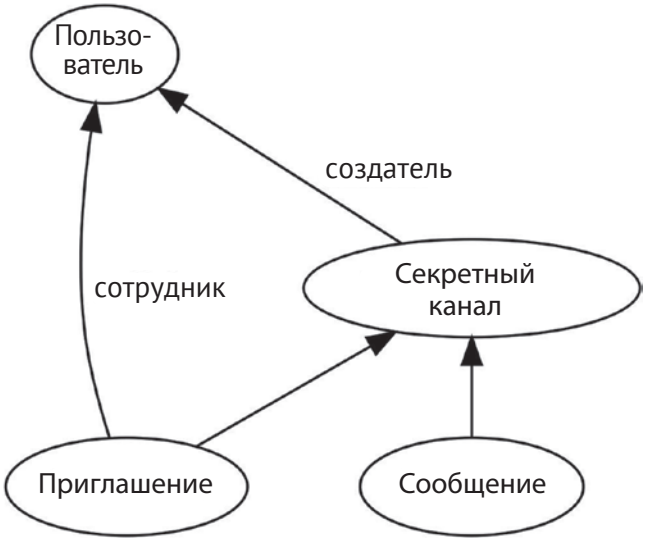


**Рис. 11.9.** Секретный канал – это группа, в которую приглашаются сотрудники

Если бы мы создавали периодически подключаемое мобильное приложение или PWA для этой модели, «Секретный канал» был бы отличным выбором факта, идентифицирующего подмножество. Если пользователь приложения является создателем или сотрудником канала, то он будет ожидать, что все сообщения будут загружены на его устройство. Группа определяет подмножество графа, содержащее транзитивное замыкание своих преемников.

Этот пример иллюстрирует еще один корневой факт и еще одно правило подмножеств. Пользователь сам должен быть первым корнем. Это дает ему подмножество всех групп, которые он создал или в которые был приглашен вступить. Это подмножество, однако, должно останавливаться на корнях других подмножеств. На рис. 11.10, например, Алиса создала канал и была приглашена в канал Боба. Подмножество с Алисой в качестве корня включает ее канал

и приглашение, но не включает сообщения в ее канале. Ей специально нужно извлечь эти сообщения из данного подмножества.

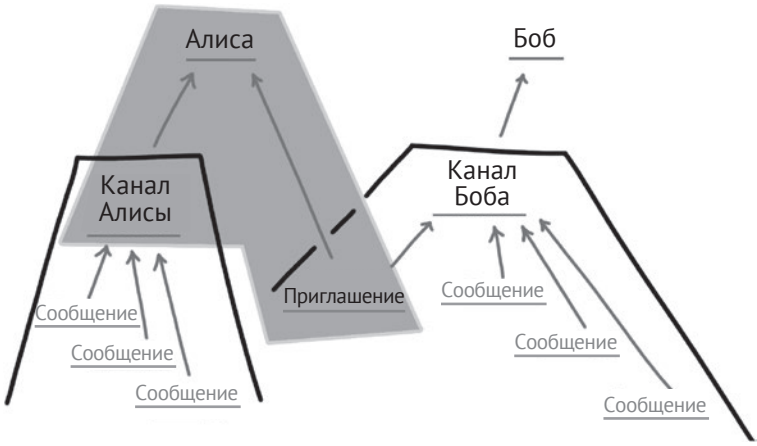


**Рис. 11.10.** Подмножество фактов под Алисой включает ее канал, но не сообщения в нем

Алгоритм идентификации фактов внутри подмножества сводится к рекурсивному обходу графа. Начните с идентифицированного корня. Рекурсивно посетите всех преемников этого факта. Добавьте преемника и транзитивное замыкание его предшественников в подмножество. Если посещенный факт не является корнем, продолжите с его преемниками. Это даст набор фактов, в которых эпизодически подключаемый клиент нуждается для обслуживания своего пользователя.

Каждый клиент хранит свой набор закладок для каждого корня. Когда он получает факты с сервера, он идентифицирует корень по хешу. Сервер отвечает пакетом фактов, которые находятся в этом подмножестве. В случае модели на рис. 11.10 первый корень – это вошедший в систему пользователь. Это позволяет получить данные о созданных им каналах и приглашениях, которые были ему отправлены. Имея подобную информацию, клиент делает дополнительные запросы для каждого из этих каналов. Это позволяет получить подмножество сообщений, которые может видеть пользователь. И каждый канал имеет свою собственную закладку.

## Предотвращение избыточных загрузок

Имея очереди для загрузки фактов и закладки для их скачивания, мы начинаем строить алгоритм для фоновой синхронизации данных в эпизодически подключаемом приложении. Но когда мы соединяем их, возникает проблема. Все факты, которые клиент выгружает, будут добавлены к их общему числу на сервере. Их будет больше, чем закладка клиента. Это означает, что они будут снова загружены клиентом при следующей загрузке. Это пустая трата пропускной способности.

Мы хотим, чтобы клиент получал только те факты, которые он сам не выгружал. Мы можем приблизиться к такому поведению, просто выполнив сначала загрузку, а затем выгрузку. Клиент загружает факты, превышающие его

текущую закладку. Он останавливается, когда выборка не возвращает никаких новых фактов. Затем он выгружает пакеты из своей исходящей очереди. Он останавливается, когда очередь пуста.

В этот момент, в идеале, единственными фактами, превышающими его закладку, будут те, которые он только что выгрузил. Поэтому если бы мы могли обновить закладку без повторной загрузки этих фактов, то избежали бы избыточности. Проблема в том, что за это время могли быть добавлены другие факты. Другие клиенты могли загрузить свои факты, или другие процессы могли создать информацию, которая должна быть отправлена клиенту. Поэтому мы не можем предположить, что сможем обновить закладку, не пропустив чего-то, что происходило параллельно.

Хорошей оптимизацией является отправка вместе с запросом на выборку списка хешей. Это факты, которые клиент только что закончил загружать. Сервер отфильтрует эти факты из ответа. В идеальном сценарии не было добавлено никаких новых фактов, и поэтому весь пакет загрузки отфильтрован. В этом случае сервер возвращает пустую коллекцию и новую закладку. Клиент обновляет свою закладку, и мы избегаем избыточных загрузок.

Если бы, с другой стороны, новые факты добавлялись параллельно, то сервер мог бы возвращать только эти новые факты. Он также вернул бы последнюю закладку, включая новые и отфильтрованные факты. Увидев этот ответ, клиент сохранит параллельные факты и обновит свою закладку. Мы добиваемся правильного поведения и избегаем избыточной загрузки.

В этом решении вся метаинформация хранится у клиента. Клиент отслеживает хеши загруженных фактов в памяти во время фоновой синхронизации. Сервер получает эту информацию в запросе и использует ее только для фильтрации ответа. Он не хранит никакой информации для каждого клиента, чтобы оптимизировать использование сети. И если что-то не получается у клиента, то он просто возвращается к правильной, пусть и неоптимальной, повторной загрузке фактов.

# Глава 12

# Генерируемое поведение

С самого начала развития индустрии программного обеспечения люди отвечали за написание программ. В каждой написанной программе принимается большое количество решений. Люди должны быть бдительными, чтобы не внедрить дефектное поведение в процессе принятия этих сложных взаимосвязанных решений.

Со временем уровень детализации, заложенный в этих решениях, поднимается до все более и более высоких абстракций. Первые разработчики программного обеспечения работали на машинном уровне. Современное программное обеспечение все чаще пишется на языках более высокого уровня. За последние несколько десятилетий управляемые среды выполнения улучшились в производительности и продуктивности настолько, что становится все труднее оправдать написание кода на аппаратном уровне. Разработчики принимают все меньше и меньше детальных решений. Продолжая подниматься по лестнице абстракции, мы можем использовать нашу новую точку обзора, чтобы взглянуть на другие острова.

Программное обеспечение пишется в виде островков поведения. База данных предназначена для хранения и запросов данных для конкретного приложения. Бизнес-логика работает с объектами в памяти. Пользовательский интерфейс представляет данные и реагирует на ввод пользователя. Сетевые API обеспечивают безопасность и коммуникации. Современная разработка приложений – это упражнение в устранении пробелов для каждого применения. Между базой данных и бизнес-логикой разработчики создают уровень доступа к данным либо вручную, либо с помощью объектно-реляционного картографа (object relational mapper – ORM). Между бизнес-логикой и пользовательским интерфейсом разработчики пишут модели представления. А между бизнес-логикой и сетью они разрабатывают пользовательские контроллеры, действия и прокси для пользовательских API.

Неизменяемая архитектура дает нам возможность изменить эту ситуацию. Вместо пользовательских островков поведения мы можем построить среду выполнения из независимых от приложения компонентов. Неизменяемая среда выполнения устранит пробелы с помощью сгенерированных моделей поведения. Мы уже показали, как могут работать некоторые из этих общих компонентов. Общее хранилище данных хранит факты, объявленные на языке фактологического моделирования, и выполняет сгенерированные конвейеры

запросов. Эти же конвейеры автоматически инвертируются, чтобы проинформировать пользовательский интерфейс, когда новый факт поступает от сетевого партнера. Правила авторизации определяют, каким из входящих фактов можно доверять и хранить их.

С каждым техническим решением, которое мы автоматизируем, мы соединяем островки системы, которые когда-то разрабатывались отдельно. Осталось устранить лишь пару пробелов. Во-первых, нам нужно спроецировать результаты запроса на пользовательский интерфейс. Во-вторых, сервер должен определить, какими фактами поделиться с партнером. Эти два пробела можно устранить с помощью двух концепций – проекции и заинтересованности. Как только мы устраним эти пробелы, мы перейдем от чисто человеческой парадигмы разработки к той, которая поддерживает высокоуровневые рассуждения. Мы получим не только производительность и уверенность, но и новую степень общения и сотрудничества.

# Проекции

Запросы в языке фактологического моделирования создают наборы фактов. Хотя эти наборы результатов полезны для разработчика, они недоступны в сыром виде для пользователя. Неизменяемая среда выполнения должна преобразовывать необработанные факты в объекты, которые пользователь может более легко воспринимать. Чтобы преодолеть разрыв между результатами запроса и пользовательским интерфейсом, мы определяем *проекцию*.

Проекция – это функция, которая отображает группу наборов фактов в структуру объектов. Внешняя среда может использовать ее для создания пользовательского интерфейса. Функциональная структура, такая как React, выразит компоненты пользовательского интерфейса как функции структуры объекта. Структура привязки данных, например XAML, будет использовать объектную структуру в качестве модели представления. В любом случае, структура начинается с проекции фактов, а не с самих фактов.

Проекции повышают уровень детализации. Они объединяют несколько небольших разрозненных фактов в целостное представление. В результате одного запроса получается набор фактов, все они одного и того же типа. Это могут быть все сущности, которые должны появиться в списке, или все возможные значения одного свойства. Пользовательский интерфейс, однако, должен будет объединить несколько наборов результатов. Чтобы создать список, ему понадобится не только набор сущностей, но и возможные значения нескольких их свойств.

## Определение проекций

Рассмотрим пример, когда администратор ресторана просматривает набор свободных столиков. В пределах этого списка он хочет увидеть номер столика, вместимость и имя официанта, закрепленного за столиком. Чтобы получить список столиков, приложение выполнит следующий запрос:

```
query tablesAvailable(r: Restaurant) {
  match t: Table where t.restaurant = r
     such that not exists s: SeatParty where s.table = t
       such that not exists b: BusTable where b.seatParty = s
}
```

Как только приложение получит номера столиков, ему нужно будет получить их свойства. Некоторые из этих свойств неизменяемы и поэтому доступны непосредственно в факте `Table`. Например, номер столика и количество посадочных мест хранятся в факте и не меняются. Но некоторые свойства будут меняться с течением времени и поэтому будут видны только в преемниках. Для этого потребуются дополнительные запросы. Например, чтобы отобразить имя официанта, назначенного на данный момент для столика, необходимо выполнить следующий запрос для каждой сущности:

```
query serverAssignedToTable(t: Table) {
  match a: Assignment where a.table = t
       such that not exists d: AddignmentDelete where d.assignment = a
  then s: Server where s = a.server
}
```

Приложение может выполнить второй запрос для каждого результата, возвращенного от первого. Это заняло бы больше времени, чем необходимо. Такая проблема производительности широко известна как SELECT N+1. Приложение выполняет один запрос для поиска сущностей, а затем делает *N* запросов для поиска некоторого свойства каждой из этих *N* сущностей.

Было бы гораздо эффективнее объединить эти запросы в один конвейер. Это именно то, что неизменяемая среда выполнения могла бы сделать с помощью проекции. Мы можем спроецировать два предыдущих запроса в одну структуру. Такая проекция может быть представлена следующим псевдокодом:

```
projection seatingView(r: Restaurant) {
  tables: for each t: Table in tablesAvailable(r) {
     tableNumber: t.tableNumber,
     capacity: t.capacity,
     server: for each s: Server in serverAssignedToTable(t) {
       name: s.name
     }
  }
}
```

Точное выражение будет отличаться для каждого языка программирования. Например, в Jinaga, неизменяемой среде выполнения для JavaScript, проекция выглядит как объектный литерал JavaScript, где поля определяются с помощью функций. Важной частью этого псевдокода является то, что он объявляет структуру объекта на основе нескольких запросов. Если бы это выражение было оценено императивно, мы бы имели проблему SELECT N+1. Но поскольку эта проекция выражена декларативно, у нас есть возможность создать один комбинированный конвейер.

## Конвейеры проекций

Как и запрос, каждая проекция имеет начальную точку. Начальной точкой предыдущей проекции является `Restaurant`. Затем проекция определяет набор полей (в данном примере – только поле `tables`). Некоторые из этих полей являются просто функциями начальной точки. И некоторые из них, как показано выше, основаны на запросах.

Чтобы выполнить проекцию, среда выполнения строит *конвейер проекции*. Он представляет собой совокупность всех конвейеров запросов, которые появляются в полях. Он начинается с конвейеров для каждого из полей. Конвейер для запроса `tablesAvailable` показан на рис. 12.1.

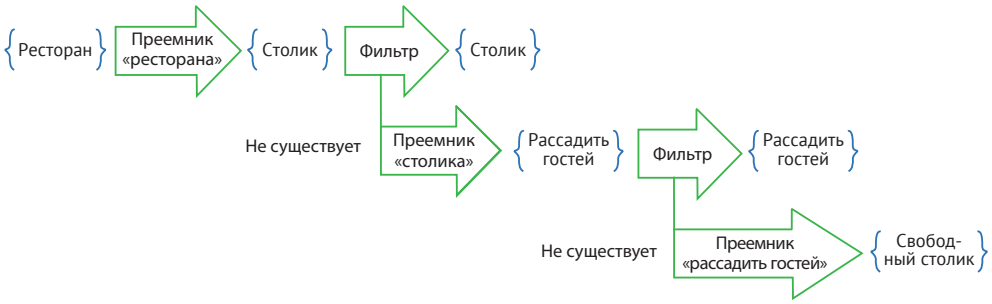


**Рис. 12.1.** Проекция начинается с конвейера для поиска столиков, которые в настоящее время свободны в ресторане

Если бы у нас было более одного поля, то мы бы начали каждый из конвейеров в одной и той же начальной точке. Если два поля производят конвейеры, которые следуют за одним и тем же первым шагом, то их конвейеры начнутся после этого общего шага. Конвейеры имеют общие шаги до того момента, когда они расходятся. Пример комбинированного конвейера показан на рис. 12.3. В нашем случае, однако, у нас есть только один конвейер, поэтому мы переходим к следующему этапу.

На следующем этапе среда выполнения рекурсивно спускается к полям, определенным для проецируемых сущностей. В этом примере сущность `table` имеет три поля – номер столика (`tableNumber`), вместимость (`capacity`) и официанта (`server`). Для двух из них выражение просто извлекает значение из факта. Для третьего, однако, задается дочерний запрос.

Для каждого дочернего запроса среда выполнения присоединяет конвейер к концу родительского конвейера. Конвейер на рис. 12.2, например, находит значения полей официанта для всех доступных столиков.

Проекционные конвейеры могут быть большими и сложными. Если бы ответственность за поддержание этих конвейеров и написание запросов к соответствующей базе данных возлагалась на человека-разработчика, он легко мог бы допустить ошибку. Но этот процесс можно автоматизировать. Начиная с декларативной проекции, фреймворк объединяет несколько конвейеров в единую составную структуру. Он вычисляет инверсию этой структуры, чтобы понять, какое именно состояние компонента или модель представления нужно обновить при поступлении нового факта. Он преобразует этот

комбинированный конвейер в запрос к базе данных для более эффективного выполнения в хранилище данных. Это позволяет избежать проблемы SELECT N+1, создавая при этом объектную модель, которая легко поддерживает пользовательский интерфейс.

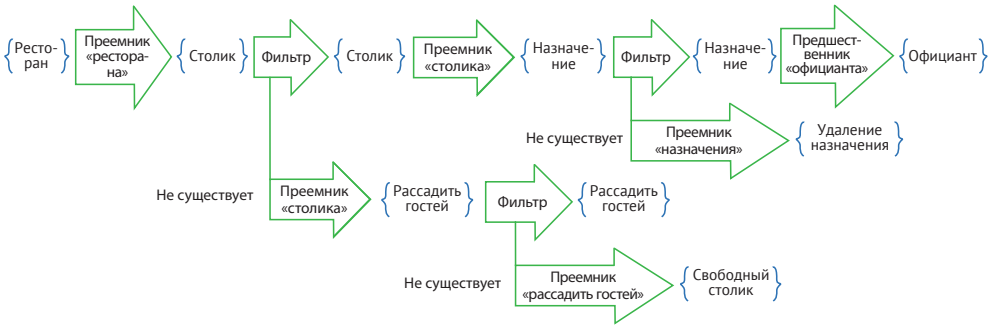


**Рис. 12.2.** Конвейер официантов присоединяется к конвейеру таблиц, чтобы найти сразу всех официантов

## Заинтересованность

Последний разрыв, который необходимо устранить, – это разрыв между бизнес-логикой и сетью. В главе 11 мы вручную решали, как делиться фактами с партнерами. Мы обнаружили опорные точки внутри модели – отношения предшественник/преемник, которые пересекают регионы. Они стали точками сотрудничества не только между пользователями, но и между узлами. Затем мы решили, как выразить это взаимодействие – очереди, темы, REST API, вебкрючки и другое. Сейчас мы рассмотрим способ автоматизировать это решение и вывести его из-под контроля человека – разработчика программного обеспечения. Мы заменим его алгоритмом для автоматического определения того, о каких фактах должен знать узел. Узлы, использующие неизменяемые среды выполнения, будут обмениваться информацией с одноранговыми узлами, чтобы выразить *заинтересованность* в фактах, которые им нужны.

Узел заинтересован в факте, если существование этого факта изменяет поведение узла. Поведение узла зависит от его конвейеров проекций и их начальных точек. Если факт влияет на результаты проецирования из ожидаемой начальной точки, то узел, который представляет эту проекцию из этой начальной точки, заинтересован в данном факте.

Чтобы выразить заинтересованность, узел делится своими проекционными конвейерами и начальными точками со своими коллегами. На основе этой информации они определяют, какими фактами следует поделиться. Итак, какие факты могут повлиять на поведение конвейера? Безусловно, начальная точка имеет большое влияние. Но, конечно, целевой узел уже знает об этом факте. Нет необходимости посылать ему факт, который он уже знает. Вместо этого узел исследует конвейер проекций, чтобы найти другие факты, которые могут повлиять на него.

Каждый шаг, который делает конвейер, посещает новый набор фактов. Если шаг находит преемников в определенной роли определенного типа, то каждый

из этих преемников может повлиять на поведение конвейера. Узел, который использует эту проекцию, заинтересован во всех этих преемниках.

Чтобы понять факт, узел должен знать обо всех его предшественниках. Когда узел выражает интерес к факту, он неявно выражает интерес к этим предшественникам. Эти факты, конечно, имеют своих предшественников. Среда выполнения вычисляет транзитивное замыкание набора интересов, чтобы найти все необходимые факты. Шаг-предшественник в рамках конвейера проекции не добавляет в набор интересов никаких фактов, которых в нем не было.

Когда конвейер включает фильтр, этот фильтр вводит дочерний конвейер. Любой факт, который влияет на дочерний конвейер, влияет на поведение фильтра. Таким образом, среда выполнения рекурсивно оценивает дочерний конвейер. За пределами фильтра конвейер продолжается. Но только факты, прошедшие фильтр, могут влиять на оставшуюся часть конвейера. И поэтому только факты, прошедшие фильтр, могут внести дальнейший вклад в набор интересов. Это приводит к некоторым интересным и противоречивым последствиям. Они наиболее очевидны в отношении и двух распространенных шаблонов, описанных в главе 8, – Delete и Period.

## Интерес к удаленным сущностям

Шаблон Delete использует факт для указания того, что объект был удален. Когда узел узнает об этом факте, он теряет интерес ко всей информации об этой сущности, кроме того факта, что она была удалена. Эту информацию необходимо сохранить.

Рассмотрим конвейер проекции на рис. 12.3. Здесь показан конвейер, необходимый для отображения всех пунктов меню, включая их название и цену. Первый фильтр конвейера исключает удаленные пункты меню. Когда он исключает удаленный пункт, узел теряет интерес к свойствам названия и цены.

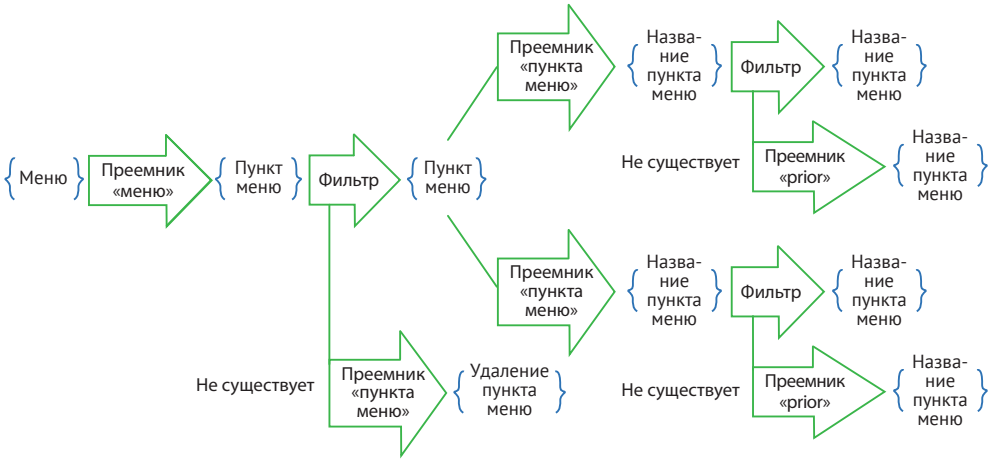


**Рис. 12.3.** Конвейер проекции, который получает названия и цены всех пунктов меню

Чтобы выполнить эту фильтрацию, узел должен знать о пунктах меню и их удалениях. Он остается постоянно заинтересованным в этих фактах. После

фильтра следуют шаги, которые исследуют свойства пункта меню. Эти шаги никогда не увидят удаленный пункт меню. Таким образом, даже если узел заинтересован в сущности и ее удалении, он не интересуется свойствами удаленных сущностей.

Это тонкая и важная оптимизация набора интересов. Вы можете подумать, что узел вообще не будет интересоваться удаленными сущностями. Как только сущность исключается фильтром удаления, то она больше не появляется в пользовательском интерфейсе. Если она не появляется, то почему узел должен быть заинтересован в ней?

Причина постоянного интереса становится более понятной, если рассмотреть, как интерес распространяется по сети узлов. Предположим, что новый узел появляется в сети и не узнает об удаленных пунктах меню. Затем предположим, что он общается с узлом, который еще не узнал, что определенный пункт меню был удален. Коллега с готовностью расскажет новому узлу о пункте меню, его названиях и ценах. Новый узел поверит, что этот пункт меню существует, и покажет его своему пользователю. Таким образом, узел, который не сохраняет интерес к удалению фактов, будет неправильно отображать удаленные сущности.

Вы могли наблюдать такое поведение в некоторых популярных движках синхронизации. Например, если вы используете Microsoft Active Directory, то можно наблюдать исчезнувшие объекты, если контроллер домена отключен за пределами времени жизни надгробия[1]. *Надгробие* в Active Directory – это запись об удалении объекта. Это аналог факта `Delete`. Active Directory не сохраняет надгробия бесконечно. Вместо этого она устанавливает время жизни от 60 до 180 дней, в зависимости от операционной системы. Если узел отключен дольше этого срока или если часы значительно дрейфовали, то его коллеги могут отбросить надгробие. Это приводит к тому, что удаленная сущность волшебным образом появляется вновь.

Единственный действительно надежный способ защиты узла от удаленных сущностей заключается в том, чтобы продолжать интересоваться фактами о сущностях и фактами их удаления. Узлу необходимо узнавать о надгробиях независимо от того, сколько им лет. Это кажется большим набором данных для хранения, но на самом деле предотвращает хранение еще большего набора данных. Сохранение полного списка надгробий не позволяет узлу заинтересоваться всеми изменениями свойств и другими действиями, связанными с этими сущностями.

Сохранение интереса к удаленным сущностям традиционно является одним из наиболее спорных результатов неизменяемых архитектур. У меня было бесчисленное количество разговоров с коллегами, которые пытались решить эту проблему с помощью правил о том, когда система может забыть о сущностях. Результаты всегда похожи на проблему «остающихся объектов», с которой сталкивается Active Directory. Если вы по-прежнему не убеждены, то, возможно, взаимодействие между интересом и шаблоном Period обеспечит более приемлемое решение. Периоды – это способ наложения времени жизни на интерес, не впадая при этом в ересь «остающихся объектов».

---

[1] Information about Lingering Objects in a Windows Server Active Directory Forest. Microsoft Knowledge Base 910205.

## Интерес к прошлым периодам

Шаблон Period описывает регулярно повторяющиеся отрезки времени, в течение которых происходят действия. Продолжительность периода – это временной интервал, который имеет смысл для приложения – дата ведения бизнеса, календарный месяц, академический семестр или финансовый год. Когда действие происходит на определенном узле, этот узел считает, что система находится в одном-единственном периоде. Он записывает действие как преемник факта периода.

Каждый узел определяет свой собственный набор интересов. Разные узлы будут проявлять интерес к разным периодам. Узлы, расположенные ближе к краю сети, обычно работают только в пределах недавних периодов. Терминал торговой точки может отображать данные только за текущую дату ведения бизнеса. Он обслуживает потребности своих пользователей в краткосрочной перспективе и оставляет долгосрочную историю серверам отчетов, расположенным ближе к центру сети.

Чтобы учесть ограничения пропускной способности и хранения данных небольших краевых устройств, мы хотим ограничить их набор интересов только самыми последними периодами. Узлы могут выбирать начальные точки своих конвейеров проецирования. Если эти конвейеры начинаются с текущего периода, то узел больше не интересуется прошлым.

Для каждого отдельного краевого узла часы периода продвигаются вперед. Вблизи от временной границы разные узлы могут считать, что система находится в разных периодах. Но это не вызовет никаких проблем при обмене фактами с более центральными узлами. Часы краевых узлов не обязательно должны совпадать с часами центральных узлов, когда центральные узлы принимают факты за пределами того периода, который они считают текущим.

Чтобы применить это решение, узел начинает свои проекции не с владельца верхнего уровня, а с периода, определенного внутри этого владельца. Например, терминал перед входом в зал в ресторане, который использует администратор, будет начинать не с факта «Ресторан» (`Restaurant`), а от преемника «Ресторанная дата ведения бизнеса» (`RestaurantDateOfBusiness`). По мере того как часы идут, терминал перед входом в зал теряет интерес к предыдущей дате ведения бизнеса. Вместо этого он запускает свои проекционные конвейеры с новой даты ведения бизнеса. Он сообщает своим коллегам начальную точку конвейера по мере изменения даты.

Эта стратегия позволяет граничным узлам ограничить свои потребности в хранении, в то время как центральные узлы берут на себя бремя хранения данных. Различные узлы сами определяют набор своих интересов. Граничные узлы выражают интерес к недавним периодам, предоставляя конвейер, начинающийся с периода. Центральные узлы выражают интерес к глубокой истории, начиная с предшественника периода.

## Совместное использование интересов

Каждый узел отвечает за определение своего собственного набора интересов. Для этого он определяет свои собственные конвейеры проекции и начальные точки. Чтобы узел мог получать факты, к которым он проявляет интерес, он должен поделиться этими конвейерами проекции и начальными точками со своими коллегами.

Независимая от приложений неизменяемая архитектура будет включать сетевой протокол для обмена конвейерами и начальными точками. Вместо того чтобы требовать от разработчиков разрабатывать индивидуальный API для конкретного приложения, она будет адаптировать свое поведение, по мере того как приложение разрабатывается. Поведение сетевого взаимодействия формируется на основе конвейеров, начальных точек, а также правил авторизации и распределения.

Узел выражает заинтересованность через сетевой протокол, не зависящий от приложения, путем предоставления своих проекционных конвейеров и начальных точек. Узел-партнер затем выполняет конвейер для поиска фактов, в которых заинтересован узел. Основываясь на постоянстве отношений с узлом, он может решить кешировать эти факты и инвертировать конвейер, чтобы сделать кеш недействительным. Или он может решить, что узел является непостоянным, как мобильное устройство, и полагаться на него для поддержания закладок, как описано в главе 11.

Начальные точки для конвейеров проекции зависят от цели, для которой служит узел. Персональные устройства, такие как мобильные устройства, как правило, начинаются с факта пользователя – человека, которому принадлежит устройство. Все остальные проекции доступны через навигацию от результатов первой. Узел объединяет все эти конвейеры вместе, чтобы создать единую структуру, исходящую из факта пользователя. Все группы, в которые вступает этот пользователь, все сущности, которые он создает, и все решения, принятые этим пользователем или для него, достижимы из этого всеобъемлющего конвейера проекции.

Узлы, занимающие центральное место в сети, обычно используются совместно многими пользователями. Они принадлежат организациям. Организация будет представлена фактом в рамках многопользовательской модели. Проекции, которые обычно выполняют такие узлы, будут создавать отчеты для всей организации, веб-сайты и общедоступные API. Их конвейеры проекций будут начинаться с факта организации.

Начальные точки будут медленно меняться, по мере того как общие узлы будут переконфигурироваться для различных арендаторов, новые пользователи будут входить в систему с личных устройств, или часы периода будут идти вперед. Проекционные конвейеры также будут медленно меняться, поскольку на каждом устройстве устанавливаются новые версии программного обеспечения. Протокол, не зависящий от приложений, позволяет каждому узлу обмениваться с партнерами информацией об этих медленных изменениях, чтобы обновить информацию о наборах фактов, в которых они заинтересованы.

## Потеря интереса

Со временем узлы теряют интерес к фактам. Когда сущность удаляется, узлы перестают интересоваться их свойствами и отношениями с другими сущностями. По мере того как часы идут, узлы теряют интерес к повседневным решениям, которые принимались в течение периода. Несмотря на то что факты неизменны и *концептуально* не могут быть удалены, для узла иногда имеет смысл *практически* удалить данные.

Узлы на границе сети, как правило, представляют собой персональные устройства. Это относительно небольшие машины с ограниченным объемом памяти, пропускной способностью и соединениями. Чтобы соблюсти их ограничения, целесообразно практически не хранить и не передавать факты, к которым они потеряли интерес.

С помощью инверсии запроса узел может распознать, когда новый факт удаляет другие факты из его набора интересов. Он может сообщить, например, что факт `Delete` привел к тому, что фильтр конвейера стал ложным и, следовательно, удалил все свойства этой сущности из своего набора интересов. Время выполнения, не зависящее от приложения, может удалить такие факты из хранилища. Однако, прежде чем применить эту стратегию, следует обратить внимание на несколько предостережений.

Во-первых, прежде чем факты будут удалены, краевой узел должен поделиться этими фактами с более постоянными партнерами. Если пользователь персонального устройства изменил свойство сущности как раз в тот момент, когда другой пользователь удалил его, то устройство может узнать об удалении до того, как у него появится шанс выгрузить изменение свойства. Если изменение свойства, которое устройство больше не интересует, удаляется из хранилища, то никакой другой узел никогда не узнает об этом. Когда удаленная сущность будет восстановлена, или будет запущен исторический отчет, решение пользователя будет потеряно.

Вторая оговорка заключается в том, что никакой другой узел не может зависеть от этого устройства как от источника фактов. Граничные устройства обычно подключаются к более центральным устройствам для получения новых фактов. Единственные факты, которые они загружают, – это те, которые пользователь устройства создал сам. Но если персональное устройство также является сервером еще более удаленного краевого устройства, то его хранилище данных не может быть очищено. Ноутбук, к которому подключено устройство-компаньон, должен сохранять историю фактов, которые могут понадобиться его спутнику, даже если он сам потерял к ним интерес.

Если инверсия запросов окажется слишком сложной или громоздкой для среды выполнения, не зависящей от приложений, существует более простая стратегия. Среда выполнения может периодически переключать одно хранилище данных в фоновый режим и начинать заполнение нового хранилища. Проекции обслуживаются из обоих хранилищ данных, фоновое обеспечивает стабильность прошлых данных, а фасадное – свежесть новых событий. Со временем все факты, интересующие узел, копируются в фасадное хранилище, а все исходящие факты удаляются из фоновой очереди. В момент затишья фоновое хранилище может быть очищено без потери информации или изменения поведения.

Какой бы ни была стратегия, важно признать разницу между практичностью очистки фактов и индуктивной строгостью сохранения интереса. Узел должен сохранять интерес к удаленным сущностям и их надгробиям, чтобы правильно защищаться от задерживающихся объектов. До тех пор, пока узел заинтересован в факте, этот факт не может быть вычищен. Только при самых строгих условиях факты могут быть удалены из хранилища. Это может произойти лишь на граничных узлах, после длительного обмена фактом с более центральным узлом и только когда интерес действительно потерян.

## Неизменяемые среды выполнения

С введением проекций и интересов последние пробелы были устранены. Теперь мы можем генерировать связи между бизнес-логикой, пользовательским интерфейсом и сетевыми коммуникациями. Генерация поведения – это не просто упрощение работы. С помощью сквозной генерации поведения мы можем обеспечить корректность приложений в такой степени, которая иначе была бы невозможна. Это дает разработчикам свободу исследовать различные варианты решений, не замыкаясь на первом подходящем. И это дает организациям автономию для внедрения инноваций в свои бизнес-процессы даже при интеграции с другими организациями.

Существует несколько способов генерировать поведение. Генерация поведения может означать генерацию кода. Некоторые генераторы кода используются в качестве шаблонов или каркасов. Как только код сгенерирован, он принадлежит разработчикам. Они несут ответственность за его изменение и поддержку. Другие генераторы кода запускаются как часть процесса сборки. Они создают файлы, которые не предназначены для изменения разработчиками. Они транспилируют[1] один язык в другой, чтобы язык, специфичный для конкретной области, мог быть легко использован языком общего назначения.

Генерация поведения может также означать изменение поведения во время выполнения для соответствия спецификации. Управляемые среды выполнения изменяют свое поведение в ответ на программы, которые они выполняют. Сборка мусора, отражение и сериализация – это сгенерированные модели поведения, которые появляются во время выполнения программы. Ни один генератор кода или транспилятор не производит программу, которая выполняет эти операции. Среда выполнения предоставляет эти услуги по мере необходимости.

Давайте представим себе решение для разработки приложений, которое использует все эти методы для генерации поведения распределенной системы. Оно будет принимать низкоуровневые решения от имени разработчика, чтобы устранить разрывы между хранением данных, бизнес-логикой, пользовательским интерфейсом, безопасностью и сетевыми коммуникациями. Оно также устраняет разрывы между приложениями, работающими на разных узлах, даже развернутыми в разное время и разработанными разными организациями. Для некоторых аспектов генерации поведения оно будет полагаться на генерацию кода, позволяя разработчикам писать непосредственно на языке моделирования фактов (Factual Modeling Language) и создавать код на предпочитаемом ими языке общего назначения. Для других аспектов оно будет использовать управляемые среды выполнения, позволяя изменять поведение без компиляции или развертывания нового кода. Целью является не быстрая разработка приложений, а корректность и автономность.

---

[1] Транспиляция – преобразование программы, при котором используется исходный код программы, написанной на одном языке программирования в качестве исходных данных, и производится эквивалентный исходный код на другом языке программирования. https://ru.wikipedia.org/wiki/Транспайлер. – *Прим. перев.*

## Генерация модели

Наш процесс разработки перспективного приложения начинается с описания проблемной области. Мы опишем сущности, свойства, действия, отношения и решения, присущие этой области. Все эти понятия моделируются как исторические факты. Инструментом, который мы будем использовать для этого описания, конечно же, является язык моделирования фактов (Factual Modeling Language).

Первое, что должна сделать среда выполнения приложения, – это перевести объявления фактов в типы данных на языке общего назначения. Лучше всего это сделать с помощью генератора кода, работающего непрерывно в процессе сборки. Выходом этого генератора кода является исходный код, чтобы его можно было понять в интегрированной среде разработки (integrated development environment – IDE) и при проверке типов. Но он не предназначен для того, чтобы разработчики вносили в него изменения после генерации.

Определяющей характеристикой фактов является то, что они неизменяемы. Несколько языков программирования общего назначения поддерживают неизменяемые структуры данных, но большинство из них по умолчанию допускают возможности их изменения. Генератор кода должен использовать соответствующие языковые идиомы, чтобы препятствовать модификации объектов, представляющих факты. В языках с сильной семантикой защиты, таких как Java и C#, это почти полностью возможно. Но в таких языках, как JavaScript, со слабой или отсутствующей защитой, разработчик просто обязан не изменять сгенерированные структуры данных.

Сгенерированный код должен сохранять структуру фактов. Он должен быть способен вычислять канонический хеш факта из свойств объекта. Он должен помогать определить, когда факт уже существует и когда он создается впервые. Он должен понимать отношения предшественник/преемник, чтобы разработчики могли формулировать запросы. Некоторые из этих поведений могут быть записаны в коде, некоторые могут быть извлечены с помощью отражения, а другие требуют, чтобы сгенерированный код сохранял структурные детали во время выполнения.

## Выполнение запросов

После того как разработчик определил всего несколько уровней модели данных, он может приступить к написанию запросов. В цикле разработки приложения запросы предоставляют первый шанс для разработчика проверить свои гипотезы. Они выражают структуру сущностей и решений в терминах фактов с верой в то, что эти факты будут поддерживать требования готового программного обеспечения. Если разработчики могут написать запрос, который удовлетворяет требованию, то у них появляется больше уверенности в том, что модель корректна.

Крайне важно, чтобы у разработчика был способ быстро итерировать модель и набор запросов, особенно на ранних этапах разработки приложения. Полная неизменяемая среда выполнения должна включать игровую площадку, на которой разработчики могут определять факты, создавать экземпляры, а затем выполнять запросы. Подумайте об этом как об эквиваленте инструмента проектирования SQL, такого как MySQL Workbench, pgAdmin или Microsoft SQL

Server Management Studio (SSMS). Это исследовательский и диагностический инструмент для разработчика, а не для приложения.

Для того чтобы приложение могло выполнить запрос, необходима среда выполнения приложения. Среда выполнения переводит запрос, выраженный в Factual или на исходном языке, в конвейер. После этого конвейер может быть выполнен на любом количестве хранилищ данных, вставлен в проекционные конвейеры или инвертирован. Конвейер – это просто внутренняя структура данных неизменяемой среды выполнения, которой можно манипулировать в различных целях.

В некоторых воплощениях нашей перспективной системы разработки приложений генератор кода превращает запросы Factual в описание конвейера, хранящееся в коде. Он генерирует функцию, которая является входной точкой для выполнения этого конвейера, обеспечивая при этом необходимые преобразования типов, чтобы гарантировать, что функция берет соответствующую начальную точку и выдает соответствующий набор фактов. В других воплощениях среда выполнения способна преобразовать запрос на исходном языке, например .NET Linq, в конвейер. Она тщательно проверяет, что запрос не выходит за пределы набора возможностей намеренно ограниченного Factual-запроса.

## Тестирование

По мере того как разработчик приложения итерирует модель и запросы, он захочет закодировать свои ожидания в повторяющихся тестах. Автоматизированное тестирование уже давно является краеугольным камнем гибкой разработки приложений, даже став движущей силой при разработке приложений на основе тестирования (test-driven development – TDD). В цикле разработки приложений, о котором здесь идет речь, тестирование играет меньшую роль. Тесты не определяют дизайн и даже не доказывают правильность. Они просто проверяют нашу работу.

Когда люди отвечают за разработку внутренних и внешних компонентов программных систем, они неизбежно совершают ошибки. Эти ошибки приводят к дефектному поведению. Мы называем их «багами» (bugs) и делаем вид, что они неизбежны. Но, как мы уже неоднократно видели на протяжении этой книги, можно доказать теоремы о поведении программного обеспечения. Если люди разрабатывают код уровня доступа к данным, сервисы бизнес-логики и модели представлений для пользовательского интерфейса, то они могут внести ошибки. Но если неизменяемая среда выполнения вычисляет инверсии запросов в соответствии с математически доказанными правилами, то мы можем быть уверены, что пользовательский интерфейс будет корректно обновляться при изменении состояния. Тестирование сгенерированного поведения среды выполнения не имеет большой ценности. Тесты должны быть направлены на проверку замысла программы, а не только ее реализации.

Разработчик пишет тесты на предпочтительном языке общего назначения, используя сгенерированный код и неизменяемую среду выполнения. Тесты выполняются с хранилищем данных, не зависящим от приложения, которое работает полностью в памяти. Это почти наверняка будет отличаться от постоянного хранилища данных, используемого в реальной системе. Но разработчик не тестирует правильность выполнения запросов в разных хранилищах данных. Он проверяет соответствие написанного запроса своему замыслу.

## Взаимодействие с пользователем

После нескольких итераций над моделью, запросами и тестами разработчик достигает точки, в которой он уверен, что описанная система соответствует намеченным требованиям. В этот момент он может начать отображать это описание в пользовательском интерфейсе. Пользовательский интерфейс развивался параллельно, в виде серии эскизов. Теперь он становится дополнен Factual-запросами и математическими формулами. Когда у нас есть и описание предполагаемого пользовательского интерфейса, и описания модели и запросов, мы можем наконец начать объединять их.

Разработчик объявляет проекцию на своем языке общего назначения. Проекции включают в себя запросы, которые генератор кода преобразовал в функции. Проекция смешивает эти функции с инструментами агрегирования и вычисления, которые предоставляет сам язык. В Linq, например, сгенерированная функция может возвращать IQueryable. Это представляет собой сам запрос, а не его результаты. Возможности языка, уже встроенные в Linq, могут составлять потоки IQueryable, сопоставлять их с помощью лямбда-выражений[1] и других собственных функций и построить сложный объект, идеально подходящий для поддержки пользовательского интерфейса.

В зависимости от целевой платформы пользовательского интерфейса неизменяемая среда выполнения обеспечивает дополнительную поддержку. В приложении на расширяемом языке разметки приложений (Extensible Application Markup Language – XAML) среда выполнения реализует событие INotifyPropertyChanged, для того чтобы перейти от инверсии запроса к привязке данных. В приложении React среда выполнения хранит проецируемые данные в состоянии компонента и предоставляет крючки для эффективного доступа. Независимо от выбранной внешней технологии, разработчик работает на уровне декларативной проекции, основанной на запросах к неизменяемым фактам.

Наконец, разработчик реагирует на события ввода. Когда пользователь нажимает на кнопку или изменяет поле, разработчик преобразует это действие в новый факт. Среда выполнения осуществляет все необходимые действия для преодоления разрыва между пользовательским интерфейсом и другими частями системы. Она сохраняет факт с помощью хранилища данных, не зависящего от приложения. Факт ставится в очередь, чтобы быть переданным более центральным узлам. Она определяет заинтересованность в распространении факта среди партнеров. И выполняет инверсии запросов, чтобы определить, какие другие части пользовательского интерфейса были затронуты. Человек-разработчик больше не отвечает ни за одно из этих технических решений.

Взаимодействие с пользователем может не принимать форму пользовательского интерфейса, который отображается на веб-странице или мобильном устройстве. Это может быть API, потребляемый некоммутативными системами. Эти методы по-прежнему применимы, просто с некоторыми технически-

[1] LINQ – это технология quering (Language Integrated Query). LINQ широко использует lambda в качестве аргументов стандартных методов оператора запроса, таких как предложение Where. lambda – это анонимная функция, содержащая выражения и операторы. https://coderoad.wiki/11344019/.

ми изменениями. URL-адреса обозначают начальные точки. Проекции отображают результаты запросов к структурам данных. Эти структуры данных затем переводятся в JSON, XML или другой транспортный формат. А такие операции, как `POST` и `PUT`, создают новые факты, которые представляют создание и обновление сущностей. Самое большое различие заключается в том, что API, как правило, меньше зависят от обновлений в реальном времени, чем пользовательские интерфейсы. И поэтому инверсии играют меньшую роль – они используются для аннулирования кеша, а не для продвижения обновлений.

Разработчики могут решить реализовать еще один уровень тестов на этом этапе. Они протестировали запросы и модель, которая использовалась для проекций. Теперь у них есть возможность протестировать сами проекции. Они могут пожелать, чтобы при нажатии пользователем определенной клавиши в определенном списке появились данные. Это намерение может тестироваться на проекции – модели представления или состояния компонента, а не на самом пользовательском интерфейсе. Это дает разработчикам инструмент для проверки сценариев взаимодействия без зависимости от более изменчивых компонентов пользовательского интерфейса. И эти тесты можно выполнять с использованием хранилища данных в памяти без потери достоверности.

## Взаимодействие

Последним этапом цикла разработки перспективного приложения является определение взаимодействия между пользователями, между системами и между узлами. Поведение одного субъекта будет влиять на поведение, наблюдаемое другим субъектом. Мы выражаем эти точки взаимодействия в неизменяемом приложении, используя интерес.

Разработчик приложения регистрирует все свои конвейеры проекции в среде выполнения. Это представляет собой намерение, чтобы приложение выражало интерес к каждому факту, необходимому для обновления его интерфейса. Разработчики могут воспользоваться этой возможностью, чтобы оптимизировать проекции для более точного вычисления интересов, или предоставить дополнительные проекции, поддерживающие другие функции приложения. Но положение по умолчанию, при котором регистрируется набор проекций, используемых в пользовательском интерфейсе, должно давать разумные результаты.

Далее, разработчики приложений вызывают интерес с определенных начальных точек. Например, когда пользователь входит в систему, он сообщает среде выполнения, что набор проекций, начинающихся с этого пользователя, стал активным. Это заставит среду выполнения начать извлекать связанные факты, так что проекции будут обновляться по мере их поступления. Если приложение включает в себя навигацию от одного представления к другому, разработчику предстоит сделать выбор. Он может присоединить проекции последующих экранов к концу проекции основного экрана. Таким образом, факты будут доступны до того, как пользователь перейдет к ним. Или же он может подождать, пока произойдет навигация, минимизируя потребляемую каналом полосу пропускания до тех пор, пока пользователь не решит, что посмотреть. Любой из этих вариантов легко выразить с помощью нескольких вызовов функций в неизменяемой среде выполнения.

Среда выполнения, со своей стороны, делится этими конвейерами проекций со своими коллегами. По мере того как приложение запрашивает выполнение конвейеров для заданной начальной точки, среда выполнения посылает эту сериализованную начальную точку и составленные конвейеры интересов. Коллеги выполняют конвейеры для определения фактов в наборе интересов. Они также выполняют транзитивное замыкание, чтобы найти всех предшественников этих фактов. Наконец, они используют закладки, чтобы договориться о том, какие факты уже известны сверстнику, а какими необходимо поделиться. По мере обмена фактами коллеги обновляют свои закладки. Все это достигается без дополнительных указаний со стороны разработчика.

Для разработчиков это наиболее естественное время для введения авторизации и правил распределения. Неизменяемые среды выполнения выполняют правила авторизации при получении нового факта, чтобы определить для себя, стоит ли доверять ему и хранить его. Среда выполнения выполняет правила распределения при отправке фактов, чтобы отфильтровать данные, на которые у коллеги нет разрешения на просмотр, даже если он проявил интерес. Эти правила можно было бы сформулировать раньше, но совместная работа дает разработчикам прекрасную возможность проверить их.

Те немногие решения, которые *были* приняты разработчиком, – как объединять проекции, когда переходить к новым начальным точкам, а также правила авторизации и распределения, заслуживают нового этапа тестирования. Разработчик пишет набор тестов, которые показывают, что когда один пользователь выполняет действие на одном устройстве, другой пользователь видит определенный эффект на другом. Они демонстрируют, что защищенные данные не видны одноранговым пользователям и что несанкционированные действия будут отклонены. Эти тесты пишутся для проекций, так как это выражения, которые наиболее точно соответствуют реальному опыту пользователя. Но они включают в себя поведение кода, который регистрирует конвейеры интересов, начальные точки и правила безопасности. Во время тестирования среда выполнения использует не только хранилище данных в памяти, но и имитатор внутрипроцессного взаимодействия. Ему не нужно устанавливать взаимосвязанные узлы или использовать ненадежную инфраструктуру во время теста. Этот имитатор не будет применяться в рабочем приложении, но он позволяет протестировать важные решения, которые разработчик принимает на этом этапе. Корректное поведение реальной коммуникационной системы регулируется математикой бесконфликтных реплицированных типов данных (CRDTs) и покрывается набором тестов поставщика среды выполнения.

Разработчики повторяют эти фазы, наращивая модель, составляя новые запросы, конструируют новые проекции и обеспечивают более сложное взаимодействие. На протяжении всего процесса решения, которые они принимают на текущей фазе, согласованы с теми, которые они принимали в прошлом. Инструменты обеспечивают эту согласованность посредством генерации кода и проверки типов. Разработчики создают рабочее приложение в течение нескольких итераций. Разработчик ни в коем случае не может выразить одно намерение уровню доступа к данным, а другое – сетевой подсистеме. Он не может вводить ошибки, основанные на разногласиях между островками поведения. Все поведение генерируется, будь то транспонированным кодом или

интерпретируемой средой выполнения, из выражения намерения. Это дает разработчикам уверенность в исследовании новых бизнес-решений и автономию для опробования новых идей.

## Неизменяемые организации

По мере того как неизменяемые архитектуры получают признание, они будут начинаться как эксперименты в рамках отдельных организаций. Разработчики приложений будут опробовать концепции, шаблоны и техники, описанные в этой книге. Они будут внедрять структуры, обеспечивающие неизменяемые среды выполнения, такие как проекты с открытым исходным кодом Correspondence и Jinaga, которые поддерживает автор. По мере того как они будут добиваться успеха, неизменяемость найдет свой путь в другие бизнес-подразделения организации.

Распространение будет постепенным. Когда бизнес-подразделениям понадобится интегрироваться друг с другом, они будут поставлены перед выбором. Принять ли нам неизменяемое взаимодействие между сервисами или мы используем более традиционные простой протокол доступа к объектам (Simple Object Access Protocol – SOAP), передачу состояния представления (Representational State Transfer – REST) или сервисную шину предприятия (Enterprise Service Bus – ESB)? Бизнес-подразделения, которые медленно внедряют неизменяемость, обнаружат, что им необходимо реализовывать дополнительные уровни адаптеров, и не получат преимущества конечной согласованности и разрешения конфликтов. Но те, кто начнет внедрять новый стиль разработки приложений, вскоре обнаружат его преимущества в быстром и надежном исследовании и экспериментировании.

### Основа для принятия решений

Бизнес-подразделения, а не отделы, станут движущей силой внедрения неизменяемости. Бизнес-подразделение включает в себя все функции, необходимые для создания ценности в рамках одного этапа потока создания ценности. Для того чтобы эффективно работать, подразделение должно быстро внедрять инновации, чтобы реагировать на меняющиеся требования. Если оно сможет воспользоваться изменениями в окружающей среде, то сможет продемонстрировать ценность для всей организации. Если не сможет, то его функции могут быть переданы на аутсорсинг. В условиях такого давления бизнес-подразделения в прошлом переходили на облачные платформы, в то время как ИТ-отдел организации оставался позади с локальными сетями. Это же давление будет побуждать их к принятию более исследовательских и автономных архитектур.

В конце концов, основой организации станет распределенный обмен неизменяемыми данными. Бизнес-подразделения будут самостоятельно разрабатывать решения на этой основе. Некоторые из них будут ценными, а другие отомрут. Но те, которые останутся, станут источником надежных бизнес-данных для других экспериментов. Возникшая неизменяемая структура данных станет инфраструктурой принятия решений, на которой организация строит свой поток ценностей.

Безусловно, в краткосрочной перспективе (и, скорее всего, в долгосрочной тоже) неизменяемые приложения будут работать бок о бок со статическими приложениями. Для этого потребуются точки интеграции. Используя шаблон Outbox, веб-крючки и эмулированную изменяемость, разработчики неизменяемых приложений найдут способы устранить эти пробелы. Со временем они будут опубликовывать эти интеграции в качестве многократно используемых компонентов для других разработчиков неизменяемых приложений. Точки интеграции станут небольшими неизменяемыми приложениями, живущими сами по себе в рамках архитектуры, прокси для внешних сервисов.

Организации, которые развивают неизменяемую платформу для принятия решений, окажутся в более выгодном положении, чем их статичные коллеги. У них будет надежный журнал всех бизнес-решений, которые были приняты для создания ценностей в компании. Они будут обмениваться информацией безопасно и свободно через границы бизнес-подразделений. И у них будет свобода для внедрения инноваций в новые решения без накладных расходов, связанных с гегемонией крупных предприятий.

## Глобально распределенные системы

Насколько мощной может быть неизменяемая архитектура в рамках одного приложения или одной организации, настолько же возрастают преимущества, когда организации с неизменяемой архитектурой интегрируются друг с другом. Открывая друг друга, они прорвут завесу эмулированной мутации и откроют скрытые под ней неизменные факты. Они предоставят доступ к своим протоколам синхронизации данных, чтобы две организации могли объединить свои неизменяемые модели.

По мере распространения неизменяемости от организации к организации границы безопасности останутся прочно закрепленными. Правила авторизации и распределения будут по-прежнему контролировать доступ к конфиденциальным фактам. Асимметричные ключи будут так же идентифицировать отдельных участников, а их цифровые подписи и общие ключи будут продолжать защищать данные в состоянии покоя и при проходе через недоверенные узлы. Вопросы безопасности будут решаться на логических границах между регионами, а не только на физических границах сетей и с помощью частных API.

Организации будут ограничивать связь между своими моделями, сохраняя точки интеграции высоко в причинно-следственной цепи. Они будут согласовывать предшественников, и каждая будет создавать свои собственные факты-преемники. Они будут развивать эти модели, используя структурное управление версиями, чтобы различные партнеры могли развивать интеграции в своих собственных направлениях без центральной координации.

По мере того как пары неизменяемых организаций объединятся в кластеры, они будут формировать сети взаимосвязанных моделей, общих для всех участников. Они будут основываться на общих сетевых протоколах и протоколах безопасности, не зависящих от приложений, и каждая будет поддерживать свои собственные хранилища данных. Из этих кластеров возникнет общая глобальная распределенная система, обменивающаяся неизменяемыми записями в безопасно интегрированной истории.

В прошлые годы подобный вид зарождающейся глобальной сети показался бы нелепым. Но мы уже видели два успешных примера. Интернет – это нецентрализованный управляемый набор серверов. Это система открытых стандартов, внедряемых огромным количеством поставщиков и управляемая свободным консорциумом сотрудничающих организаций. И блокчейн не является центральной базой данных, которая полагается на физическую и сетевую безопасность для контроля доступа. Это публичные распределенные реестры неизменяемых записей, выполняемые на различных узлах, работающих на конкурирующих реализациях открытого стандарта.

Основываясь на этих примерах, я уверен, что глобальная неизменяемая сеть появится. У нее не будет изменяемого мышления REST API, которое привязывает их к конкретному местоположению. Она также не будет иметь неэффективности доказательств работы, привязывающих ее к линейной истории. Вместо этого она будет применять результаты десятилетий математических исследований для создания действительно глобально распределенных систем, которые не зависят от местоположения, конвергентны, безопасны и автономны.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

**Майкл Л. Перрии**

**Искусство неизменяемой архитектуры: теория и практика управления данными в распределенных системах**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Минц С. В.*
Научный редактор *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 31,53. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Прочитав эту книгу, вы поймете преимущества использования неизменяемых объектов в ваших распределенных системах. Вы узнаете набор правил для идентификации и обмена неизменяемыми объектами, а также увидите коллекцию полезных теорем, которые гарантируют, что распределенные системы, которые вы строите, будут иметь конечную согласованность. Используя шаблоны, вы найдете, где истина сходится, увидите, как изменения происходят ассоциативно, а не последовательно, и придете к комфортному пониманию того, что нет единственного источника истины. Приведенные практические примеры наглядно показывают, как создавать программное обеспечение, используя описанные шаблоны, методы и инструменты. К концу чтения вы будете владеть языком и ресурсами, необходимыми для уверенного анализа и построения распределенных систем.

*Книга предназначена для архитекторов программного обеспечения и опытных разработчиков. Она содержит примеры на SQL и таких языках, как JavaScript и C#. Полезен опыт работы с распределенными вычислениями, моделированием данных или бизнес-анализом.*

**Вы научитесь:**

- оценивать распределенную систему с точки зрения неизменяемых объектов;
- распознавать проблемы в существующих проектах и вносить небольшие изменения для их устранения;
- создавать новую систему с нуля, применяя шаблоны;
- применять принципы неизменяемой архитектуры к своим инструментам, включая базы данных SQL, очереди сообщений и сетевые протоколы, которые вы уже используете;
- открывать для себя новые инструменты, в которых эти принципы применяются изначально.

**Интернет-магазин:**
www.dmkpress.com

**Оптовая продажа:**
КТК «Галактика»
books@alians-kniga.ru

Apress

ДМК издательство
www.дмк.рф

ISBN 978-5-93700-111-5