Рефакторинг баз данных

эволюционное проектирование

Refactoring Databases

Evolutionary Database Design

Scott W. Ambler
Pramodkumar J. Sadalage

ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokio • Singapore • Mexico City

Рефакторинг баз данных

эволюционное проектирование

Скотт В. Эмблер
Прамодкумар Дж. Садаладж

Москва • Санкт-Петербург • Киев
2007

ББК 32.973.26-018.2.75
С14
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Эмблер, Скотт В., **Садаладж**, Прамодкумар Дж.

С14 Рефакторинг баз данных: эволюционное проектирование. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2007. — 672 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1157-5 (рус.)

В настоящей книге приведены рекомендации, касающиеся того, как использовать методы рефакторинга для усовершенствования баз данных. Она посвящена описанию процедур проектирования базы данных с точки зрения архитектора объектно-ориентированного программного обеспечения, поэтому представляет интерес и для разработчиков прикладного кода, и для специалистов в области реляционных баз данных. В книгу включены многочисленные советы и рекомендации по улучшению качества проектирования базы данных. Значительное место уделено описанию того, как действовать в тех практических ситуациях, когда база данных уже существует, но плохо спроектирована, или когда реализация первоначального проекта базы данных не позволила получить качественную модель. Прежде всего книгу можно использовать в качестве технического руководства для разработчиков, непосредственно занятых на производстве. С другой стороны, она представляет собой теоретическую работу, стимулирующую дальнейшие исследования в направлении объединения объектно-ориентированного и реляционного подходов.

ББК 32.973.26-018.2.75

ISBN 978-8459-1157-5 (рус.)
ISBN 0-321-29353-3 (англ.)

Оглавление

Содержание

Посвящается Беверли, моей прекрасной молодой невесте.

Скотт

Посвящаю женщинам, которых я люблю больше всего, Рупали и нашей дочери Аруле.

Прамод

Об авторах

Скотт В. Эмблер — консультант в области усовершенствования программных процессов (Software Process Improvement — SPI), проживающий недалеко от Торонто. Скотт создал методологии AM (Agile Modeling) (`www.agilemodeling.com`), AD (Agile Data) (`www.agiledata.org`), EUP (Enterprise Unified Process) (`www.enterpriseunified-process.com`) и AUP (Agile Unified Process) (`www.ambysoft.com/unifiedprocess`), а также возглавил работы по их усовершенствованию. Скотт — автор нескольких книг, в том числе *Agile Modeling* (John Wiley & Sons, 2002), *Agile Database Techniques* (John Wiley & Sons, 2003), *The Object Primer 3rd Edition* (Cambridge University Press, 2004), *The Enterprise Unified Process* (Prentice Hall, 2005) и *The Elements of UML 2.0 Style* (Cambridge University Press, 2005). Скотт работает в качестве редактора с правами публикации в коммерческом журнале *Software Development* (`www.sdmagazine.com`), участвует и выступает с программными речами на различных международных конференциях, включая *Software Development*, *UML World*, *Object Expo*, *Java Expo* и *Application Development*. Скотт закончил Университет Торонто, получив степень магистра информатики. В свободное время Скотт изучает стили каратэ — годзюрю и кобудо.

Прамодкумар Дж. Садаладже — консультант компании ThoughtWorks, которая занимается разработкой и интеграцией приложений для предприятия. Прамод стал инициатором внедрения методов и процессов эволюционного проектирования и рефакторинга баз данных в 1999 году, работая над крупными приложением J2EE с использованием методологии экстремального программирования (Extreme Programming — XP). С тех пор Прамод, принимая участие во многих проектах, неизменно применяет эти методы и процессы. Прамод охотно делится своими знаниями в области администрирования баз данных в эволюционных проектах, применения эволюционных процессов к базам данных и влияния эволюционных подходов на администрирование базы данных, и в письменном, и в устном виде, чтобы дать возможность всем желающим освоить эволюционные методы проектирования баз данных. Свое свободное время Прамод проводит с женой и дочерью, совершая прогулки и бегая трусцой.

Предисловия

Ниже в данном разделе приведены предисловия, написанные авторитетными специалистами.

- Десятилетие назад термин “рефакторинг”[1] был известен лишь ограниченному числу профессионалов, главным образом принадлежащих к сообществу пользователей языка Smalltalk. Но было очень интересно наблюдать за тем, как постепенно увеличивается количество желающих использовать рефакторинг в качестве формального и эффективного способа модификации работающего кода. В конечном итоге в наши дни многие рассматривают рефакторинг кода как важную составляющую разработки программного обеспечения.

  Сферой моей деятельности является разработка приложений для предприятий, причем значительная часть такой разработки связана с обеспечением взаимодействия с базами данных. Впервые выпустив книгу по рефакторингу, я отметил базы данных (БД) как основную проблемную область при проведении рефакторинга, поскольку при попытке применить операции рефакторинга к БД приходится сталкиваться с новым кругом проблем. Острота этих проблем усугубляется тем заслуживающим сожаления барьером, который обнаруживается в сфере разработки программного обеспечения для предприятий, поскольку специалисты в области баз данных и разработчики программного обеспечения отделены друг от друга стеной взаимного непонимания и недовольства.

  В работе Скотта и Прамода мне особенно понравилось то, что они стремятся каждый по-своему найти точки соприкосновения между этими категориями специалистов и объединить их усилия. Скотт неизменно пытается преодолеть указанный разрыв, подготавливая все новые и новые книги по базам данных, а написанное им по объектно-реляционному отображению оказало большое влияние на мои собственные сочинения, посвященные архитектуре приложений предприятия. Прамод, возможно, менее известен, но оказал на мои взгляды не меньшее влияние. Когда он начинал работу вместе со мной над одним проектом в компании ThoughtWorks, нам говорили, что проведение операций рефакторинга баз данных — невыполнимая задача. Прамод не прислушался к этим предостережениям, взял за основу некоторые наметки идей и превратил их в формализованную программу, позволяющую привести схему базы данных в постоянное, но управляемое движение. Благодаря этому разработчики приложений также получили возможность использовать методы эволюционного проектирования в своем коде. С тех пор Прамод опробовал эти методы на предприятиях многих наших клиентов, распространил их среди наших коллег из компании ThoughtWorks, поэтому, по край-

[1] Интересной аналогией, которая могла бы способствовать усвоению понятия “рефакторинг”, является то, что одно из толкований английского слова refactoring — перестановка мест слагаемых. — *Примеч. пер.*

ней мере для нас, базы данных навсегда исключены из списка объектов, не поддающихся развитию на основе непрерывного проектирования.

В этой книге сконцентрирован опыт двух человек, которые научились выживать на ничейной земле между приложениями и данными, а также приведены рекомендации, касающиеся того, как использовать методы рефакторинга для усовершенствования баз данных. Специалисты, знакомые с таким подходом, как рефакторинг, смогут отметить такую важную особенность описанных методов, что они позволяют обеспечить непрерывный перенос самих данных, а не только модификацию программ и структур данных. В этой книге показано, как достичь такой цели, и все, что в ней описано, подтверждается опытом проектирования (и положительным, и отрицательным), который был накоплен этими двумя специалистами.

Несмотря на то что я в восторге от появления этой книги, я также надеюсь, что это — только первый шаг. Меня очень радует то, что после выхода в свет моей книги по рефакторингу появилось много новых сложных инструментальных средств, позволяющих автоматизировать значительную часть операций рефакторинга. Я надеюсь, что то же самое произойдет применительно к базам данных, и мы увидим, как поставщики инструментальных средств предлагают инструменты, позволяющие упростить задачу непрерывного переноса схемы и данных для каждого. Но прежде чем это произойдет, вы сможете воспользоваться данной книгой, чтобы создать собственные процессы и инструменты для упрощения своей работы, а в дальнейшем эта книга будет оставаться для вас ценным источником знаний, позволяющим успешно использовать указанные инструментальные средства рефакторинга баз данных.

Мартин Фаулер, редактор серии; руководитель исследовательских работ компании ThoughtWorks

- С того времени, когда я начал свою карьеру в области разработки программного обеспечения, изменились до неузнаваемости многие аспекты производства и технологии. Но не изменилось одно — фундаментальный характер разработки программного обеспечения. Задача создания хотя бы какого-то программного обеспечения никогда не была особенно сложной — достаточно лишь получить в свое распоряжение компьютер и начать выдавать “на-гора” код. Но создать хорошее программное обеспечение всегда было трудно, и тем более было очень нелегко разработать превосходное программное обеспечение. Эта ситуация не изменилась и сегодня. В наши дни стало удобнее создавать все более крупные и сложные программные системы, собирая воедино фрагменты программ, полученных из различных источников, возможности инструментальных средств разработки программного обеспечения еще больше расширились, и мы гораздо лучше стали разбираться в том, какие подходы к осуществлению процесса разработки программного обеспечения принесут или не принесут результаты. Тем не менее основная часть программного обеспечения все еще продолжает оставаться ненадежной, а борьба за достижение допустимых уровней качества кажется нескончаемой. Возможно, причиной такого положения дел является то, что создаваемые системы становятся все крупнее и сложнее, или, возможно, то, что в используемых методиках все еще не преодолены какие-то фундаментальные недостатки. Я счи-

таю, что разработка качественного программного обеспечения и в наши дни остается столь же трудоемкой, как и прежде, из-за совместного проявления этих двух факторов. К счастью, время от времени появляются новые технологии и методики, внушающие надежду. Среди этих достижений редко встречаются новинки, которые позволяли бы резко повысить нашу способность реализовать тот потенциал, который внушал нам надежду при запуске большинства проектов. Методы, применяемые в рефакторинге, наряду со связанными с ними адаптивными методологиями, относятся именно к таким редкостным достижениям. Работа, результаты которой изложены в этой книге, расширяют возможности появления перспективных технологических достижений в очень важном направлении.

Рефакторинг — это контролируемый метод безопасного улучшения проекта кода без изменения его функциональной семантики. Возможность добиться усовершенствования кода дана любому разработчику, но подход, основанный на проведении рефакторинга, предусматривает применение формального метода безопасного внесения изменений (с помощью тестов) и распространение полученных при этом знаний, накопленных сообществом по разработке программного обеспечения (в виде формальных определений операций рефакторинга). С того времени, как была издана оригинальная книга Фаулера по этой теме, нашли широкое распространение операции рефакторинга и были приняты на вооружение инструментальные средства, позволяющие обнаружить потенциальные возможности проведения операций рефакторинга и применить эти операции к коду. Но применительно к тому уровню в многоуровневой структуре приложений, на котором используются данные, оказалось, что проведение операций рефакторинга является намного более затруднительным. Как показано в этой книге, связанные с этим проблемы отчасти обусловлены применением сложившихся ранее подходов, но не менее важно то, что еще не был четко определен процесс и набор операций рефакторинга, применимых к уровню данных. Этот фактор действительно оказывал неблагоприятное воздействие, поскольку низкое качество проекта на уровне данных почти всегда становится причиной возникновения проблем на более высоких уровнях, как правило, вызывая применение целого ряда неудачных проектных решений, принятых для осуществления бесполезных усилий стабилизировать шаткий фундамент. Более того, невозможность обеспечить развитие уровня данных, то ли из-за отрицания необходимости в этом, то ли из-за страха перед изменениями, препятствует реализации способностей всех остальных участников разработки создать наилучшее возможное программное обеспечение. Работа, проделанная авторами, столь важна потому, что позволяет решить именно эти проблемы; мы теперь имеем организационные и технологические возможности итеративного внесения усовершенствований в проекты, применяемые в этой жизненно важной области.

Меня очень обрадовало появление данной книги, и я надеюсь, что она станет стимулом к созданию инструментальных средств поддержки описанных в ней методик. В настоящее время наблюдается интересный этап развития индустрии программного обеспечения, связанный с постоянно растущей значимостью программного обеспечения с открытым исходным кодом и тех средств совместной разработки кода, которые с ним связаны. Такие проекты, как Eclipse Data Tools Platform, становятся естественным центром притяжения для всех тех разработчи-

ков, которые стремятся воплотить в жизнь методы рефакторинга базы данных, создавая соответствующие инструментальные средства. Я надеюсь, что сообщество программистов, участвующих в разработках с открытым исходным кодом, внесет свой вклад в реализацию этого замысла, поскольку потенциальный выигрыш очень велик. Разработка программного обеспечения перейдет на качественно новый уровень, после того как операции рефакторинга базы данных станут такими же распространенными и широко применяемыми, как и обычные операции рефакторинга.

Джон Грэм (John Graham), компания Eclipse Data Tools Platform, руководитель проекта, председатель комитета, старший инженер по кадрам, компания Sybase, Inc.

- Сообщество специалистов в области обработки данных во многом осталось в стороне от тех ярких достижений, которые стали результатом внедрения адаптивных методов разработки программного обеспечения. Безусловно, разработчики приложений приняли на вооружение подход, предусматривающий применение рефакторинга, проведение разработки на основе тестирования и других подобных методов, позволяющий повысить производительность разработки программного обеспечения и достичь других существенных преимуществ, а специалисты в области обработки данных в основном проигнорировали эти тенденции и даже отстранились от них.

  Это стало для меня очевидно в самом начале моей карьеры в качестве разработчика приложений в крупном учреждении, предоставляющем финансовые услуги. В то время кабинка, в которой находилось мое рабочее место, располагалась непосредственно между территориями, занимаемыми группой разработчиков и группой специалистов по базам данных. Мне не потребовалось много времени, чтобы понять, что эти две группы, расстояние между рабочими местами которых не превышало нескольких метров, весьма различаются по своим подходам, методам и применяемым ими процессам. Поступление любого запроса клиента в группу разработчиков влекло за собой проведение определенного рефакторинга, осуществление входного контроля кода и выполнение всеобъемлющего приемочного тестирования. А при поступлении аналогичного запроса в группу специалистов по базам данных происходил запуск формального процесса утверждения изменений, который требовал прохождения через несколько уровней организационной структуры еще до того, как могла начаться модификация схемы. Такой процесс был чрезвычайно трудоемким и неизбежно вызывал раздражение и разработчиков, и клиентов, но оставался в прежнем виде, поскольку группа специалистов по базам данных не знала какого-либо иного пути.

  Но им приходится искать другой путь, если они хотят, чтобы их деловое предприятие смогло выжить в современной обстановке, характеризующейся постоянным обострением конкурентной борьбы. В частности, сообщество специалистов по базам данных должно найти какой-то способ овладения адаптивными методами, которые применяют их коллеги-разработчики.

  Книга *Рефакторинг баз данных: эволюционное проектирование* — это бесценный источник информации, который показывает специалистам в области обработки данных, как именно они могут освоить новые методы и приступить к уверенному, безо-

пасному осуществлению изменений. В своей книге Скотт и Прамод показали, как усовершенствовать проект с помощью небольших, итеративных операций рефакторинга, позволяющих администратору базы данных, используя адаптивные методы, избежать ошибок, неизменно возникающих при осуществлении крупных, заранее подготовленных проектов, и обеспечить развитие схемы наряду с приложением, по мере того, как будет постепенно достигаться лучшее понимание требований клиента.

Но не следует заблуждаться — задача проведения рефакторинга баз данных остается сложной. Даже такое простое изменение, как переименование столбца, каскадно распространяется по всей схеме, влияя на объекты схемы, инфраструктуры доступа к базе данных и прикладной уровень, поэтому администраторы базы данных стремятся избегать любых изменений.

В книге *Рефакторинг баз данных: эволюционное проектирование* приведено описание набора рекомендуемых методов, которое позволяет профессиональному администратору базы данных подробно узнать о том, как перенести адаптивные методы в среду проектирования и разработки баз данных. Скотт и Прамод уделяют внимание мельчайшим подробностям того, что нужно сделать, чтобы фактически реализовать каждую методику рефакторинга базы данных, демонстрируя осуществимость операций рефакторинга и открывая путь к их широкому распространению.

Это может послужить призывом к действию для всех специалистов в области обработки данных. Изучайте эти методы, берите их на вооружение и делитесь своим опытом с коллегами. Рефакторинг баз данных — это ключ к внедрению адаптивных методик в проблемной области обработки данных.

Сакин Рекхи (Sachin Rekhi), руководитель программы, Microsoft Corporation

- Разработчики, участвующие в создании прикладного программного обеспечения, руководствуются двумя основными подходами: разработчики приложений главным образом используют объектно-ориентированные (ОО) средства, широко применяют язык Java и адаптивные методы разработки программного обеспечения, а специалисты в области реляционных баз данных прежде всего стремятся провести всесторонний технический анализ и создать надежный проект реляционной базы данных. Эти две категории специалистов используют разные языки программирования, посещают разные конференции, и даже, кажется, почти не поддерживают отношения друг с другом. Такие взаимоотношения между специалистами обнаруживаются в отделах информационной технологии многих организаций. Разработчики объектно-ориентированного программного обеспечения жалуются, что администраторы баз данных — закоренелые консерваторы, неспособные успевать за быстрыми темпами изменений. А специалисты в области баз данных безнадежно машут рукой, вспоминая примитивные ошибки разработчиков программ на языке Java, не имеющих ни малейшего представления о том, что делать с базой данных.

  Скотт Эмблер и Прамод Садаладже принадлежат к той редкой группе людей, которые способны указать путь к объединению усилий всех категорий специалистов в области программирования. Книга *Рефакторинг баз данных: эволюционное проектирование* посвящена описанию процедур проектирования базы данных с точки зрения архитектора объектно-ориентированного программного обеспече-

ния. Поэтому настоящая книга представляет ценность и для разработчиков объектно-ориентированного программного обеспечения, и для специалистов в области реляционных баз данных. Она поможет разработчикам объектно-ориентированного программного обеспечения научиться применять адаптивные методы рефакторинга кода для работы с базами данных, а также позволит специалистам в области реляционных баз данных получить представление о том, как осуществляется создание объектно-ориентированного программного обеспечения.

В книгу включены многочисленные советы и рекомендации по улучшению качества проектирования базы данных. Она в основном сосредоточена на описании того, как действовать в тех практических ситуациях, когда база данных уже существует, но плохо спроектирована, или когда реализация первоначального проекта базы данных не позволила получить качественную модель.

Эта книга обречена на успех, поскольку в ней удалось достичь сразу нескольких целей. Прежде всего ее можно использовать в качестве технического руководства для разработчиков, непосредственно занятых на производстве. С другой стороны, эта книга представляет собой теоретическую работу, стимулирующую дальнейшие исследования в направлении объединения объектно-ориентированного и реляционного подходов. Я хотел бы, чтобы системные архитекторы прислушались к призывам Эмблера и Садаладже признать, что базы данных могут трактоваться гораздо шире, чем просто место для хранения перманентных копий классов.

Д-р Пол Дорси (Paul Dorsey), президент компании Dulcian, Inc.; президент нью-йоркской группы пользователей Oracle; председатель группы J2EE SIG.

Введение

За последние несколько лет в индустрии информационной технологии (ИТ) стремительно распространились эволюционные методологии разработки программного обеспечения, часто называемые адаптивными, такие как экстремальное программирование (Extreme Programming — XP), метод Scrum, унифицированный процесс компании Rational (Rational Unified Process — RUP), адаптивный унифицированный процесс (Agile Unified Process — AUP) и разработка на основе функций (Feature-Driven Development — FDD). Чтобы было проще понять особенности этих методов, подчеркнем, что эволюционный метод по своему характеру является и итеративным, и инкрементным, а адаптивный подход является эволюционным и вместе с тем характеризуется высокой степенью взаимодействия участников разработки. Кроме того, в организациях, применяющих информационные технологии, все шире внедряются такие адаптивные методики, как рефакторинг, программирование в паре, разработка на основе тестирования (Test-Driven Development — TDD) и адаптивное проектирование на основе модели (Agile Model-Driven Development — AMDD). Эти подходы и методики были разработаны и получили свое развитие в течение многих лет в среде рядовых разработчиков и доведены до совершенства обычными программистами, а не придуманы теоретиками, живущими в башнях из слоновой кости. Короче говоря, эти эволюционные и адаптивные методики, по-видимому, невероятно успешно действуют на практике.

В своей оригинальной книге *Refactoring* Мартин Фаулер дал определение *рефакторинга* как небольшого изменения в исходном коде, которое способствует улучшению проекта кода без изменения его семантики. Иными словами, рефакторинг — это улучшение качества сделанной вами работы без нарушения или добавления чего-либо. Кроме того, в своей книге Мартин обсуждает идею, что если возможно подвергнуть рефакторингу прикладной исходный код, то есть возможность подвергнуть рефакторингу схему базы данных. Но Мартин указал, что рефакторинг баз данных — очень сложная задача, поскольку базы данных отличаются высокой степенью связности; поэтому он решил исключить эту тематику из своей книги.

После публикации книги *Refactoring* в 1999 году оба автора настоящей книги стали искать способы проведения рефакторинга схем базы данных. Первоначально мы работали отдельно, встречаясь друг с другом на таких конференциях, как *Software Development* (`www.sdexpo.com`), и ведя переписку с помощью списков рассылки (например, `www.agiledata.org/feedback.html`). Мы обсуждали идеи, посещали лекции и презентации друг друга на конференциях и вскоре обнаружили, что наши идеи и методы пересекаются, а также являются весьма взаимосовместимыми. Поэтому мы объединили свои усилия в написании настоящей книги, чтобы поделиться своим опытом и рассказать о методах развития схем баз данных путем проведения операций рефакторинга.

Все примеры, приведенные в книге, написаны на языке Java и на языках баз данных Oracle. Практически каждое описание операции рефакторинга базы данных включает код, предназначенный для модификации непосредственно самой схемы базы данных, а

применительно к некоторым наиболее интересным операциям рефакторинга мы показали, какое влияние они могут оказать на прикладной код Java. Безусловно, нельзя найти две базы данных, которые были бы неотличимыми друг от друга, поэтому мы включили описания альтернативных стратегий реализации в тех случаях, когда между программными продуктами баз данных имеются тонкие, но важные различия. В некоторых случаях мы обсуждаем альтернативные реализации каких-то аспектов рефакторинга с помощью таких характерных для СУБД Oracle средств, как команды `SET UNUSED` или `RENAME TO`, а во многих приведенных нами примерах кода используются средства `COMMENT ON`, предусмотренное в СУБД Oracle. В других программных продуктах баз данных предусмотрены другие средства, позволяющие упростить рефакторинг баз данных, и хороший администратор базы данных должен знать, как воспользоваться этими особенностями в своих интересах. Но лучше всего, если в будущем появятся инструментальные средства рефакторинга баз данных, которые могли бы выполнять всю эту работу за нас. Кроме того, мы стремились показать настолько простой код Java, чтобы можно было преобразовать его в код C#, C++ или даже Visual Basic практически без затруднений.

Необходимость в осуществлении эволюционной разработки баз данных

Настало такое время, что эволюционные подходы к разработке баз данных должны выйти на передний план. Это означает, что схема базы данных не должна полностью проектироваться заранее еще до начала проекта; вместо этого схема наращивается на протяжении всего периода осуществления проекта, отражая изменения в требованиях, которые выдвигают лица, заинтересованные в разработке проекта. Нравится нам это или нет, но по мере развития проекта требования изменяются. В традиционных подходах принято отрицать эту фундаментальную истину и пытаться “управлять изменениями”; термин *управление изменениями*, кроме прочих отрицательных оттенков, несет смысл — создание препятствий для изменений. Вместо этого специалисты, непосредственно использующие современные методы разработки, решили принять потребность в изменениях как должное и взять на вооружение методы, позволяющие им совершенствовать плоды своего труда настолько оперативно, насколько это согласуется с расширяющимися требованиями. Программисты стали руководствоваться такими методами, как TDD, рефакторинг и AMDD, а также создали новые инструментальные средства разработки, чтобы упростить использование этих методов. Достигнув этой цели, специалисты в области программного обеспечения осознали, что нужны также методы и инструментальные средства для поддержки эволюционной разработки баз данных.

Преимущества эволюционного подхода к разработке баз данных включают следующее.

1. **Минимизация бесполезных затрат.** Эволюционный, своевременный (Just-In-Time — JIT) подход позволяет исключить издержки, которые неизбежно возникают при использовании последовательных методов в случае изменения требований. Все инвестиции, заблаговременно вложенные в подготовку детализированных требований, создание архитектуры и проектирование артефактов, становятся напрасно потерянными, если в дальнейшем обнаруживается, что требование, ради выполнения которого были сделаны эти затраты, больше не выдвигается. Ведь если ваша квалификация позволяет сделать работу заранее, то,

очевидно, вы сможете воспользоваться своей квалификацией, чтобы сделать ту же работу, когда для этого настанет время.

2. **Предотвращение необходимости в существенных переделках.** Как будет показано в главе 1 "Эволюционная разработка базы данных", все равно должно быть заранее проведено некоторое заблаговременное начальное проектирование, позволяющее продумать наперед основные проблемы и выяснить, какие сложности могли бы потенциально привести к существенным переработкам при их обнаружении на более поздних этапах проекта; не нужно лишь предварительно изучать мельчайшие подробности.
3. **Постоянная уверенность в наличии работоспособной системы.** Эволюционный подход позволяет регулярно выпускать рабочее программное обеспечение (даже если его развертывание осуществляется только в демонстрационной среде), которое всегда может быть передано в эксплуатацию. Если в вашем распоряжении каждый раз через одну-две недели оказывается новая, работоспособная версия системы, риск неудачного завершения проекта резко уменьшается.
4. **Постоянная уверенность в том, что существующий на данный момент проект базы данных имеет наивысшее возможное качество.** Именно в этом состоит вся суть подхода, основанного на проведении рефакторинга баз данных: усовершенствование проекта схемы на основе постепенно осуществляемых небольших изменений.
5. **Применение подхода к разработке, совместимого с подходом других разработчиков.** Разработчики программного обеспечения руководствуются эволюционным подходом, и если специалисты в области обработки данных хотят стать равноправными членами современных групп разработчиков, они также должны выбрать для своей работы один из эволюционных методов.
6. **Сокращение общей трудоемкости.** Применяя в своей производственной деятельности эволюционный подход, вы выполняете только ту работу, которая фактически нужна сегодня, и ни на йоту больше.

Тем не менее при проведении разработки базы данных на основе эволюционных методов необходимо учитывать некоторые возникающие при этом сложности, которые описаны ниже.

1. **Наличие разных подходов у различных категорий разработчиков.** Многие специалисты в области обработки данных предпочитают придерживаться последовательного подхода к разработке программного обеспечения и часто продолжают утверждать, что до начала программирования должны быть созданы в той или иной форме детализированные логические и физические модели данных, которые должны быть взяты за основу. После перехода к использованию современных методологий приходится отказываться от этого подхода, который теперь рассматривается как слишком неэффективный и рискованный; в связи с этим многие специалисты в области обработки данных чувствуют себя растерянными. Но хуже всего то, что многие "идейные руководители" в сообществе специалистов по базам данных — это люди, чье профессиональное становление произошло в 1970-х и 1980-х годах, но для них осталась незамеченной объектная революция, происшедшая в 1990-х годах, поэтому они не

смогли своевременно приобрести опыт в эволюционных разработках. Мир изменился, но эта категория людей, по-видимому, не хочет изменяться вместе с ним. Как описано в данной книге, специалисты в области обработки данных не только имеют возможность организовать свою работу по-новому, на основе эволюционных или даже адаптивных методов, но и сами эти методы фактически являются предпочтительным способом организации работы.

2. **Низкая скорость обучения.** Для изучения описанных в данной книге методов потребуется время, но еще больше времени займет полная перестройка сложившихся последовательных подходов и переход к использованию эволюционных подходов.
3. **Отсутствие полностью сложившихся инструментальных средств поддержки.** Ко времени публикации книги *Refactoring* в 1999 году отсутствовали какие-либо инструментальные средства, поддерживающие эту методику. Но всего лишь через несколько лет в каждой отдельной интегрированной среде разработки (Integrated Development Environment — IDE) появились непосредственно встроенные функции рефакторинга кода. Ко времени написания этой книги также не существовали какие-либо инструментальные средства рефакторинга баз данных, но авторы фактически включили весь код, необходимый для реализации операций рефакторинга вручную. К счастью, в проспекте проектов Eclipse Data Tools Project (DTP) указано на необходимость разработки функциональных средств рефакторинга баз данных в составе интегрированной среды разработки Eclipse, поэтому остается лишь дождаться того, что поставщики инструментальных средств подхватят это начинание.

Суть адаптивных методов

Безусловно, данная книга не посвящена непосредственно адаптивной разработке программного обеспечения, но нельзя отрицать того факта, что основным методом для разработчиков, руководствующихся адаптивным подходом, должен стать рефакторинг баз данных. Процесс рассматривается как адаптивный, если он соответствует четырем критериям, разработанным организацией Agile Alliance (`www.agilealliance.org`). Эти критерии определяют предпочтения, а не альтернативы, побуждая стремление сосредоточиваться на определенных областях, но не исключать другие. Иными словами, если для вас представляют ценность одни сравниваемые концепции, то вы должны еще больше ценить другие концепции, рассматриваемые в сравнении. Например, нельзя отрицать важность процессов и инструментальных средств, но индивидуумы и способы взаимодействия между ними еще важнее. Четыре адаптивных критерия описаны ниже.

1. **Индивидуумы и способы их взаимодействия ВАЖНЕЕ процессов и инструментальных средств.** Наиболее важными аспектами, требующими размышлений, является то, какие люди участвуют в разработках и как они взаимодействуют. (Если вам не удастся добиться успехов в этой сфере, то лучшие инструментальные средства и процессы окажутся бесполезными.)
2. **Работоспособное программное обеспечение ВАЖНЕЕ всеобъемлющей документации.** Основная цель разработки программного обеспечения состоит в создании работоспособного программного обеспечения, которое отвечает требованиям

лиц, заинтересованных в появлении этого программного обеспечения. Сказанное не означает, что документация больше не нужна; документация, подготовленная должным образом, позволяет узнать, как и почему создана система, и определить, как работать с системой.

3. **Сотрудничество с клиентом ВАЖНЕЕ согласования контракта.** Только клиент может сказать вам, чего он хочет. К сожалению, в этом клиенты не очень сильны; вероятнее всего, они не обладают квалификацией, позволяющей точно определить требования к системе, не излагают правильно эти требования с самого начала, а что хуже всего, со значительной вероятностью меняют свои взгляды по поводу будущей системы с течением времени. Безусловно, важно иметь контракт со своими клиентами, но контракт не может заменить эффективное взаимодействие. Преуспевающие специалисты в области информационной технологии тесно сотрудничают со своими клиентами, не жалеют усилий, чтобы узнать, что действительно требуется их клиентам, и вместе с тем постоянно проводят обучение своих клиентов.
4. **Своевременная реакция на изменения ВАЖНЕЕ выполнения плана.** По мере осуществления разработки системы заинтересованные в этом лица начинают осознавать, что хотели бы внести изменения; обнаруживаются также изменения в деловой среде и в основополагающей технологии. В области разработки программного обеспечения изменчивость является повседневной реальностью, а это означает, что весь план проекта и общий подход могут быть эффективными лишь в том случае, если они будут отражать изменения в среде.

Как читать эту книгу

Основная часть книги, включая главы 6–11, состоит из справочного материала, в котором подробно описана каждая операция рефакторинга. В первых пяти главах изложены фундаментальные идеи и методы эволюционной разработки баз данных и, в частности, описаны принципы рефакторинга баз данных. Эти главы необходимо прочитать в следующем порядке.

- Глава 1 “Эволюционная разработка баз данных” содержит краткий обзор основных принципов эволюционной разработки и методов, которые обеспечивают такую разработку. В этой главе приведены краткие сведения о рефакторинге кода, рефакторинге баз данных, регрессионном тестировании баз данных, эволюционном моделировании данных на основе подхода AMDD, управлении конфигурациями информационных активов баз данных, а также обоснована необходимость в применении отдельных специализированных вариантов среды разработки.
- В главе 2 “Рефакторинг баз данных” подробно рассматриваются концепции, лежащие в основе рефакторинга баз данных, и показано, почему осуществление этой задачи может стать сложным на практике. В этой главе рассматривается также пример проведения рефакторинга базы данных как в простой среде с одним приложением, так и в сложной среде с несколькими приложениями.
- Глава 3 “Процесс рефакторинга баз данных” содержит подробное описание этапов, требуемых для проведения рефакторинга схемы базы данных и в простом, и в сложном варианте среды. В случае базы данных с одним приложением обеспечи-

вается гораздо больший контроль над средой, и в результате этого объем работы по проведению рефакторинга схемы намного уменьшается. В среде с несколькими приложениями необходимо предусмотреть переходный период, в течение которого в базе данных параллельно поддерживаются и старая, и новая схема, что позволяет группам разработчиков приложений обновлять и развертывать создаваемый ими код на производстве.

- В главе 4 “Развертывание на производстве” описан процесс, лежащий в основе развертывания операций рефакторинга баз данных на производстве. Такая задача может оказаться особенно затруднительной в среде с несколькими приложениями, поскольку в подобной среде приходится объединять и тестировать изменения, предлагаемые несколькими разными группами разработчиков.
- В главе 5 “Стратегии рефакторинга баз данных” подытожены некоторые из рекомендуемых методов, обнаруженных нами за многие годы, которые относятся к осуществлению рефакторинга схем базы данных. Кроме того, в этой главе изложено несколько идей, которые мы намереваемся опробовать, но еще не имели возможности этого сделать.

Благодарности

Мы хотим поблагодарить указанных ниже людей за их вклад в создание настоящей книги: Дуга Барри (Doug Barry), Гэри Эванса (Gary Evans), Мартина Фаулера (Martin Fowler), Бернарда Гудвина (Bernard Goodwin), Свена Гортса (Sven Gorts), Дэвида Хея (David Hay), Мишель Хаусли (Michelle Housely), Пола Петралиа (Paul Petralia), Майкла Терстона (Michael Thurston) и Майкла Вайздоса (Michael Vizdos).

Кроме того, Прамод хочет выразить свою признательность Срирам Нарайан (Sriram Narayan), Энди Слокаму (Andy Slocum), Ирфан Шаху (Irfan Shah), Нарайяну Раману (Narayan Raman), Анишеку Агарвалу (Anishek Agarwal) и другим своим товарищам по команде, которые постоянно оспаривали его мнение и научили многому, что касается разработки программного обеспечения. Я хочу поблагодарить также Мартина за то, что он научил меня писать, выступать и, в общем, посоветовал выйти за рамки своей деятельности в компании ThoughtWorks; Кента Бека (Kent Beck) за его поддержку; своих коллег из компании ThoughtWorks, которые во многом мне помогают и позволяют чувствовать себя на работе действительно комфортно; своих родителей Джинаппа и Шобху, которые приложили большие усилия по воспитанию меня и Правина, моего брата, и которые с раннего детства следили за тем, как я пишу, и стремились усовершенствовать мой стиль.

Ждем Ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Webсервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

Email: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем:

из России: 115419, Москва, а/я 783

из Украины: 03150, Киев, а/я 152

Глава 1

Эволюционная разработка баз данных

Водопады и каскады весьма привлекательны для туристов.
Но каскадная стратегия организации разработки проектов
программного обеспечения известна своей низкой эффективностью.

Скотт Эмблер

Все современные процессы разработки программного обеспечения, называемые также методологиями, по своему характеру являются эволюционными, поэтому в их рамках работы должны проводиться итеративно и с приращением. К примерам таких процессов относятся, в частности, Rational Unified Process (RUP), Extreme Programming (XP), Scrum, Dynamic System Development Method (DSDM), семейство методов Crystal, Team Software Process (TSP), Agile Unified Process (AUP), Enterprise Unified Process (EUP), Feature-Driven Development (FDD) и Rapid Application Development (RAD). При итеративной организации работы в каждом цикле выполняется небольшая часть общего объема действий, таких как моделирование, тестирование, разработка кода или развертывание, после чего в следующем цикле выполняется еще одна часть, затем другая и т.д. Этот процесс отличается от последовательного подхода, в котором выявляются все требования, подлежащие реализации, создается подробный проект, этот проект реализуется, выполняется проверка и, наконец, происходит развертывание готовой системы. С другой стороны, инкрементный подход предусматривает организацию работ по созданию системы в виде ряда промежуточных выпусков, а не одного основного выпуска.

Кроме того, многие из современных процессов разработки являются адаптивными, что можно кратко определить как применение эволюционного подхода и вместе с тем очень тесного сотрудничества участников разработки. В группе разработчиков, принявших на вооружение подход, предусматривающий тесное сотрудничество, происходит активный поиск способов эффективной организации совместной работы; приходится даже предпринимать попытки сделать активными участниками работы группы таких лиц, заинтересованных в разработке проекта, как клиенты делового предприятия. Кокберн [**14**] указывает, что необходимо стремиться принять на вооружение наиболее эффективную методику связи, применимую в конкретной ситуации: проведение личных встреч у доски

с мелом лучше переговоров по телефону, переговоры по телефону лучше обмена письмами по электронной почте, а переписка по электронной почте лучше, чем пересылка подробного документа. Чем теснее связь и сотрудничество в группе разработки программного обеспечения, тем выше шансы на успех.

Безусловно, и эволюционный, и адаптивный способ организации работы был с энтузиазмом принят на вооружение сообществом разработчиков программного обеспечения, но аналогичное утверждение в отношении сообщества разработчиков баз данных было бы неверным. Основная часть методологий, основанных на обработке данных, является по своему характеру последовательной и требует создания весьма детализированных моделей, поскольку лишь при таком условии можно приступать к реализации. Еще хуже то, что эти модели часто берутся за основу и над ними устанавливается контроль над внесением изменений в целях сведения изменений к минимуму. (А если рассматривать конечные результаты такого контроля, то его фактически следовало бы назвать процессом предотвращения изменений.) В этом и состоит основная проблема: общепринятые методики разработки баз данных не отражают реальностей современных процессов разработки программного обеспечения. Это положение должно измениться.

Авторы исходят из предпосылки, что специалисты в области обработки данных должны принять на вооружение эволюционные методики, аналогичные тем, которыми руководствуются разработчики прикладного программного обеспечения. Разумеется, можно было бы придерживаться той точки зрения, дескать, сами разработчики должны возвратиться к "проверенным и надежным" традиционным подходам, принятым в сообществе специалистов по базам данных, но становится все более очевидно, что традиционные методы просто перестали себя оправдывать. В главе 5 книги *Agile & Iterative Development* [**25**] Крэг Ларман собрал и подытожил результаты исследований и свидетельства подавляющей поддержки со стороны ведущих идеологов в сообществе специалистов по информационным технологиям (Information Technology — IT), которые указывают на необходимость применения эволюционных подходов. Общий итог состоит в том, что эволюционные и адаптивные методики, широко применяемые в сообществе разработчиков прикладного программного обеспечения, действуют намного успешнее по сравнению с традиционными методиками, которыми в основном пользуются представители сообщества специалистов в области баз данных.

Для специалистов в области обработки данных также открывается возможность перейти к использованию эволюционных подходов во всех аспектах своей деятельности, если они выберут такой путь дальнейшего развития. На первом шаге к этому необходимо заново продумать всю "культуру обработки данных", принятую в организации, в которой используется информационная технология (ИТ), чтобы весь подход к обработке данных соответствовал возможностям и потребностям современных групп проектировщиков, работающих в области информационной технологии. Метод применения адаптивных данных (Agile Data — AD) [**4**], который регламентирует совокупность подходов и ролей для модернизации деятельности в области применения данных, позволяет достичь именно этой цели. В подходах, принятых в методе AD, учитывается то, что данные — один из многих важных аспектов функционирования коммерческого программного обеспечения, а это способствует тому, что разработчики прикладных программ становятся более искушенными в методах обработки данных, а специалисты по базам данных начинают осваивать современные технологии и приемы разработки. Метод AD позволяет учесть то, что каждая группа проектировщиков является уникальной и должна придерживаться та-

кого процесса разработки, который в наибольшей степени соответствует сложившейся ситуации. В этом методе учитывается также то, насколько важно выйти за рамки текущего проекта и почувствовать себя причастным к решению тех проблем, с которыми сталкивается предприятие; кроме того, данный метод побуждает специалистов самого предприятия, таких как администраторы эксплуатируемых баз данных и архитекторы данных, проявлять достаточную гибкость, для того чтобы их взаимодействие с группами проектировщиков происходило динамически.

На втором шаге специалисты в области баз данных, особенно администраторы баз данных, должны принять на вооружение новые методы, позволяющие им организовать свою работу по эволюционному принципу. В настоящей главе приведен краткий обзор указанных важных методов, а также изложены доводы авторов в пользу того, что наиболее важным современным методом является рефакторинг баз данных, которому в основном посвящена эта книга. Основные методы эволюционной разработки баз данных приведены ниже.

1. **Рефакторинг баз данных.** Развитие существующей схемы базы данных путем единовременного внесения небольших изменений в целях повышения качества проекта базы данных без изменения ее семантики.
2. **Эволюционное моделирование данных.** Итеративное и инкрементное моделирование аспектов системы, связанных с применением данных, полностью аналогичное всем другим аспектам функционирования системы, для обеспечения того, чтобы схема базы данных развивалась в одном темпе с прикладным кодом.
3. **Регрессионное тестирование базы данных.** Обеспечение того, чтобы схема базы данных действительно успешно функционировала.
4. **Управление конфигурациями артефактов базы данных.** Модели данных, тесты базы данных, испытательные данные и прочие объекты являются важными артефактами проекта, которыми необходимо управлять точно так же, как и любыми другими артефактами.
5. **Специализированные варианты среды разработки.** Для разработчиков требуются собственные варианты рабочей среды, в которых они могут модифицировать часть создаваемой ими системы и добиться ее успешной работы, прежде чем объединять полученные ими результаты с результатами работы других представителей группы.

Ниже приведено подробное описание каждого эволюционного метода разработки баз данных.

1.1. Рефакторинг баз данных

Рефакторинг [**17**] — это регламентированный способ внесения небольших изменений в исходный код в целях улучшения его проекта, благодаря чему работа с этим кодом упрощается. Наиболее существенной особенностью любой операции рефакторинга является то, что она оставляет неизменной функциональную семантику кода; проведение рефакторинга не приводит ни к расширению, ни к сужению функциональных возможностей кода, поскольку в результате рефакторинга происходит лишь повышение качества

кода. В качестве примера операции рефакторинга можно указать переименование метода `getPersons()` в `getPeople()`. Для реализации такой операции рефакторинга необходимо изменить определение метода, а затем внести изменение в каждый отдельный вызов этого метода во всем прикладном коде. Операция рефакторинга может считаться законченной лишь после того, как код начинает действовать так же, как прежде.

По аналогии с этим, рефакторинг базы данных — это простое изменение в схеме базы данных, которое улучшает ее проект, сохраняя неизменной поведенческую и информационную семантику базы данных. Рефакторингу могут быть подвергнуты либо структурные аспекты схемы базы данных, такие как определения таблиц и представлений, либо функциональные аспекты, такие как хранимые процедуры и триггеры. При проведении рефакторинга схемы базы данных приходится не только переопределять саму схему, но и вносить изменения во внешние системы, такие как деловые приложения или средства выборки данных, которые привязаны к конкретной схеме. Очевидно, что задача осуществления операций рефакторинга базы данных намного сложнее по сравнению с операциями рефакторинга кода, поэтому требует особого внимания. Операции рефакторинга базы данных подробно описаны в главе 2 “Рефакторинг баз данных”, а процесс осуществления рефакторинга баз данных рассматривается в главе 3 “Процесс рефакторинга баз данных”.

1.2. Эволюционное моделирование баз данных

Безусловно, по поводу эволюционных и адаптивных методик можно услышать разные мнения, но истина заключается в том, что эти методики не основаны на старых подходах по принципу “разработки кода, а затем его исправления” под новым именем. Эти новые методологии предусматривают к тому же изучение требований, а также глубокий анализ архитектуры и проекта до начала их создания, и одним из хорошо зарекомендовавших себя способов решения этой задачи является подготовка модели до начала разработки кода. На рис. 1.1 показан жизненный цикл осуществления технологии AMDD [**3**; **5**]. Технология AMDD предусматривает создание в начале проектирования моделей высокого уровня, которые полностью охватывают всю проблемную область, затрагиваемую в процессе проектирования, а также описывают потенциальную архитектуру, лежащую в основе будущего проекта. При этом одной из обычно создаваемых моделей является “узкая” концептуальная модель проблемной области, которая отражает основные деловые сущности и связи между ними [**18**]. На рис. 1.2 приведен пример, относящийся к обычному финансовому учреждению. Степень детализации модели, рассматриваемой в данном примере, является достаточной для того, чтобы можно было приступить к созданию проекта; разработчик обязан продумать на ранних этапах проектирования лишь наиболее важные вопросы, не занимаясь непосредственно в это время проработкой ненужных деталей, поскольку все необходимые нюансы могут быть учтены позже, когда для этого настанет время.

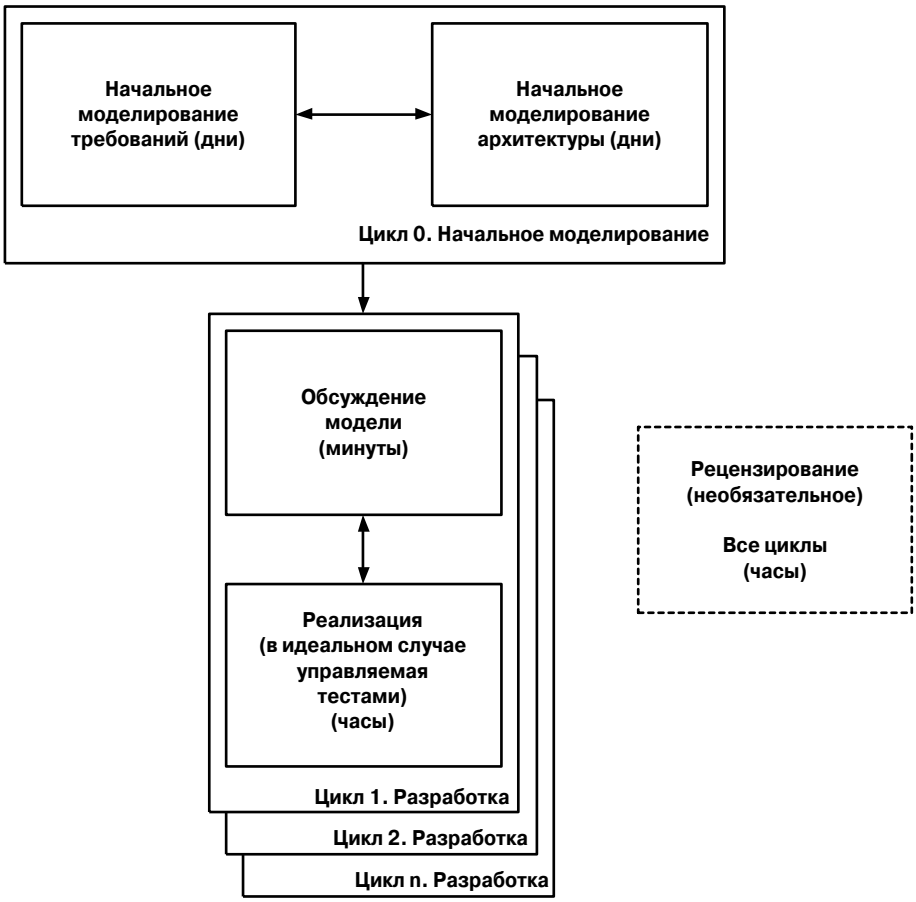


Рис. 1.1. Жизненный цикл метода AMDD (Agile Model-Driven Development — адаптивная разработка на основе модели)

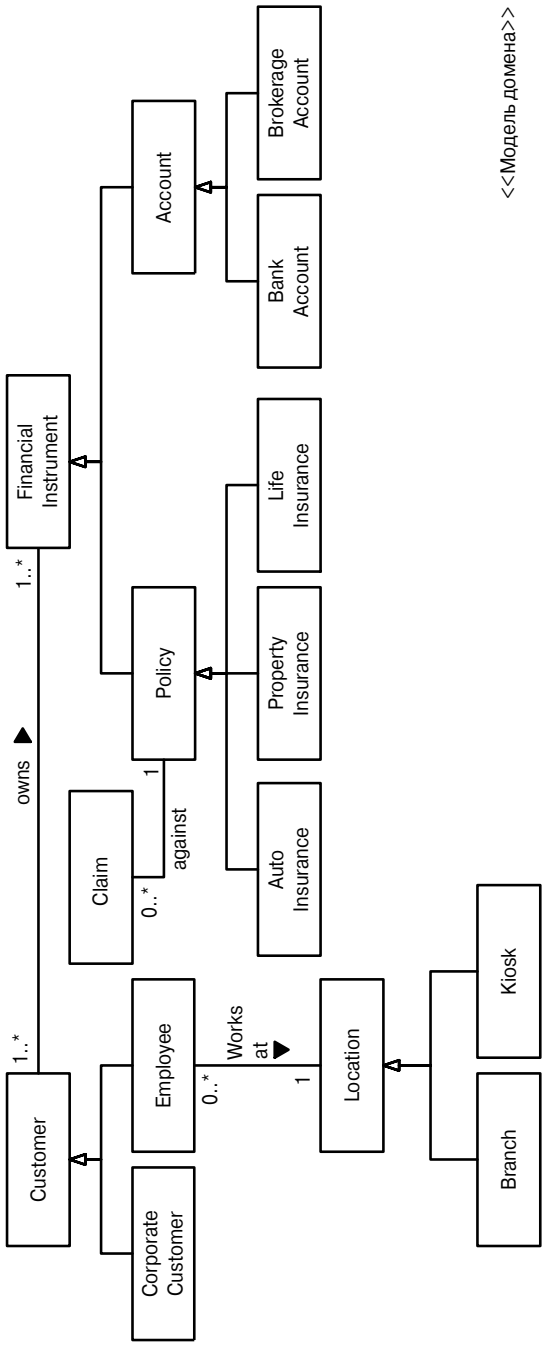


Рис. 1.2. Концептуальная модель и проблемная область, которые описывают вымышленное финансовое учреждение с помощью средств языка UML

Вполне естественно, что концептуальная модель по мере углубления знаний о проблемной области развивается, но степень детализации остается постоянной. Все нюансы отражаются в объектной модели (в качестве такой модели может рассматриваться исходный код) и в физической модели данных. Основой разработки этих моделей неизменно остается концептуальная модель проблемной области, а сама разработка проводится параллельно, наряду с другими артефактами, для обеспечения согласованности. На рис. 1.3 показана детализированная физическая модель данных (Physical Data Model — PDM), которая позволяет судить о том, каких масштабов достигает модель в конце третьего цикла разработки. Если этап, обозначенный как "цикл 0", имеет продолжительность одну неделю, то с учетом сведений о типичной продолжительности времени разработки проектов, составляющей меньше одного года, а также на основании того, что циклы разработки занимают по времени две недели, можно сделать вывод, что рассматриваемая физическая модель данных была получена в конце седьмой недели проектирования. Физическая модель данных отражает требования к данным, а также все ограничения, унаследованные от предыдущей версии, которые были обнаружены при разработке проекта вплоть до текущего момента. Требования к данным для будущих циклов разработки отражаются в моделях, создаваемых в текущих циклах, как только для этого наступает подходящий момент, т.е. по мере необходимости.

Задача эволюционного моделирования данных является сложной. Дело в том, что приходится учитывать ограничения данных, унаследованные от предыдущего проекта, но, как известно, применявшиеся ранее источники данных часто становятся причиной таких затруднений, что при недостаточном их учете разработка проекта программного обеспечения может окончиться неудачей. К счастью, квалифицированные специалисты в области баз данных хорошо понимают все особенности источников данных, применяемых в конкретной организации, и эти экспертные знания могут использоваться, если разработка ведется с учетом текущих потребностей, столь же успешно, как если бы разработка проводилась на последовательной основе. Тем не менее и при таком подходе необходимо придерживаться соглашений об интеллектуальном моделировании данных, поскольку это регламентируют стандарты прикладного моделирования, принятые в методе адаптивного моделирования. Подробный пример эволюционного/адаптивного моделирования данных приведен по адресу `www.agiledata.org/essays/agileDataModeling.html`.

1.3. Регрессионное тестирование базы данных

Для того чтобы можно было вносить изменения в существующее программное обеспечение, либо в целях осуществления рефакторинга, либо для добавления нового функционального средства, не опасаясь нарушить его работу, необходимо иметь возможность проверить то, что после проведения требуемого изменения все программы продолжают успешно функционировать. Другими словами, необходимо предусмотреть возможность выполнения в системе полного регрессионного тестирования. Если при этом будет обнаружено, что произошло нарушение работы какого-то компонента, можно либо устранить возникшее нарушение, либо выполнить откат изменений. В сообществе разработчиков прикладного программного обеспечения все чаще используется подход, предусматривающий

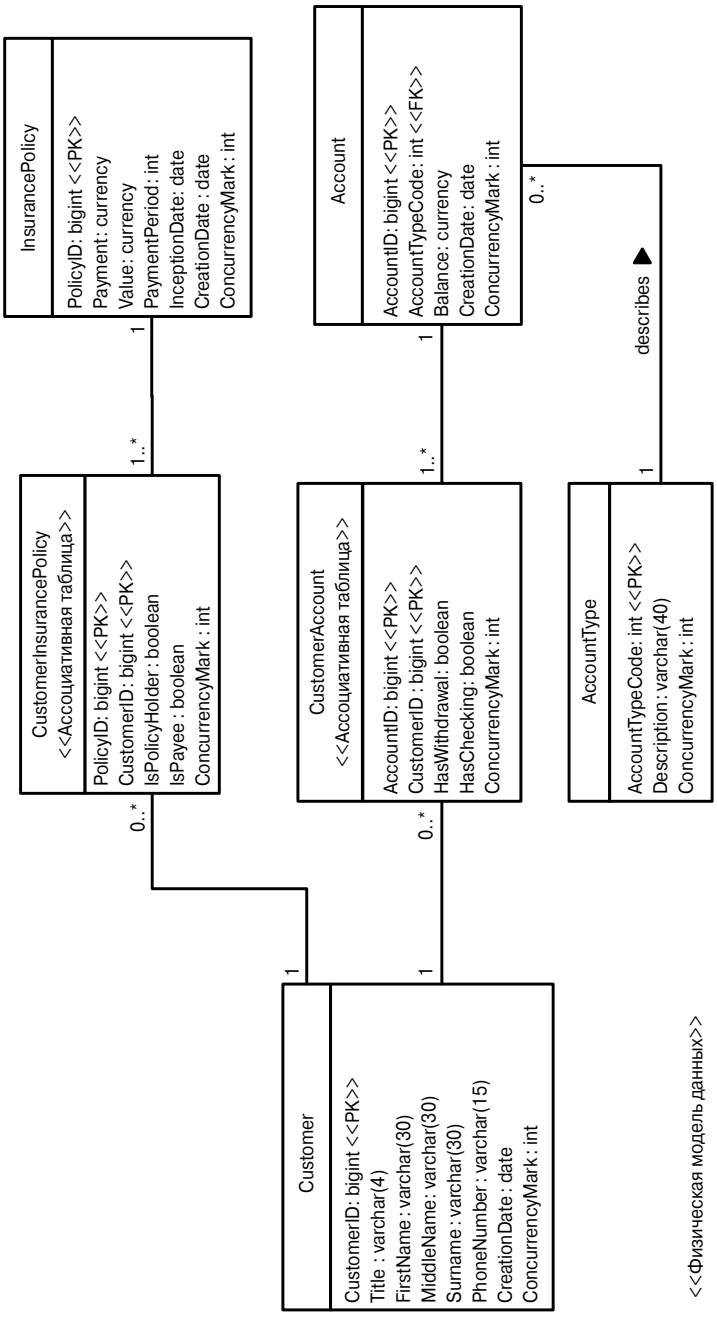


Рис. 1.3. Подробная физическая модель данных (PDM), создаваемая с помощью обозначений языка UML

разработку полного тестового набора для всех компонентов одновременно с прикладным кодом, а программисты, взявшие на вооружение методы адаптивного программирования, фактически даже предпочитают вначале написать тестовый код и только после этого приступать к созданию "настоящего" кода. Но неужели тестирование базы данных является менее важным по сравнению с тестированием прикладного исходного кода? Важная бизнес-логика реализуется в базе данных в форме хранимых процедур, правил проверки данных и правил ссылочной целостности (Referential Integrity — RI), и вполне очевидно, что эта бизнес-логика требует тщательного тестирования.

Одним из эволюционных подходов к разработке является метод создания тестов до начала разработки (Test-First Development — TFD), называемый также методом создания тестов до начала программирования (Test-First Programming — TFP); этот метод предусматривает подготовку теста, неудачное завершение которого будет указывать на наличие ошибок, еще до начала работы по разработке нового функционального кода. Как показывает блок-схема, приведенная на рис. 1.4, метод TFD состоит из нескольких этапов, описанных ниже.

1. Ускоренное создание теста, в основном с применением такого объема кода, чтобы на данный момент произошло неудачное завершение тестирования из-за отсутствия рассматриваемых функциональных средств.
2. Выполнение подготовленных тестов (чаще всего полного тестового набора, хотя в целях ускорения работы может быть решено выполнить только подмножество тестов) для проверки того, что новый тест действительно оканчивается неудачей.
3. Обновление функционального кода таким образом, чтобы он проходил новый тест.
4. Повторное выполнение всех тестов. Если тесты оканчиваются неудачей, возврат к шагу 3; в противном случае переход в начало цикла.

Основные преимущества метода TFD состоят в том, что он вынуждает тщательно продумывать все аспекты применения нового функционального средства еще до его реализации; гарантирует наличие нового тестового кода, предназначенного для проверки качества выполненной работы; а также дает уверенность, основанную на понимании того, что дальнейшее развитие системы возможно, поскольку всегда можно обнаружить, не нарушена ли работа каких-то компонентов в результате внесения изменений. Итак, наличие полного регрессионного тестового набора для прикладного исходного кода позволяет проводить рефакторинг кода, а наличие полного регрессионного тестового набора для базы данных позволяет проводить рефакторинг базы данных.

Метод разработки на основе тестов (Test-Driven Development — TDD) [**9**; **10**] представляет собой сочетание методов TFD и рефакторинга. Подход, основанный на использовании метода TFD, предусматривает вначале написание кода, а после того как код становится работоспособным, обеспечивается поддержание высокого качества проекта путем проведения рефакторинга по мере необходимости. Но вслед за осуществлением любой операции рефакторинга должны быть повторно выполнены все регрессионные тесты, а это позволяет убедиться в том, что не нарушена работа какого-либо компонента.

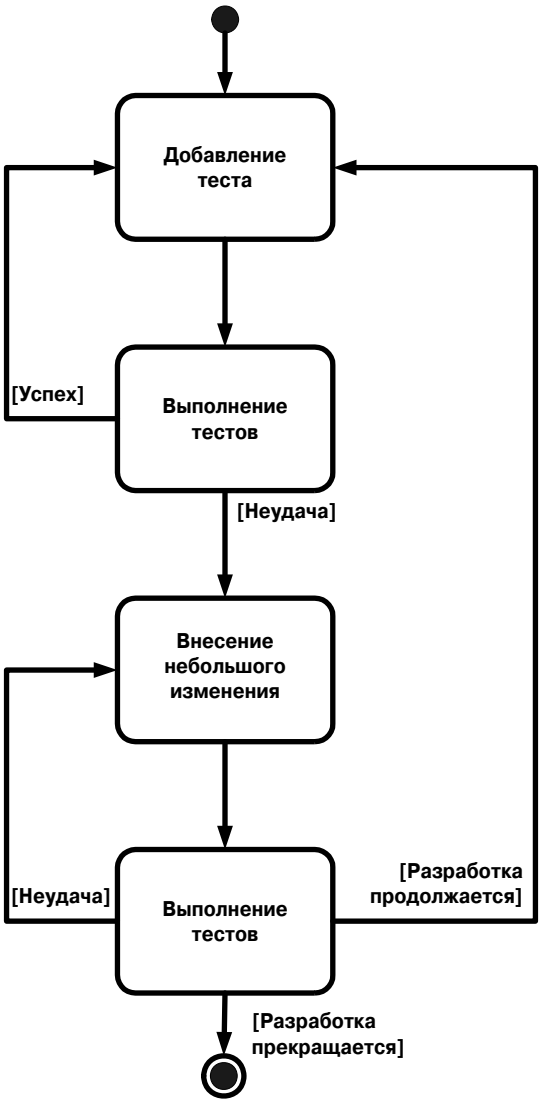


Рис. 1.4. Подход к разработке, основанный на предварительной подготовке тестов

На основании сказанного можно сделать важный вывод, что, по-видимому, необходимо будет использовать несколько инструментальных средств тестирования компонентов, по крайней мере два — один для базы данных и по одному для каждого языка программирования, используемого во внешних программах. К счастью, имеется предоставляемое бесплатно семейство инструментальных средств XUnit (например, JUnit для Java, VBUnit для Visual Basic, OUnit для Oracle), которые неплохо согласованы друг с другом.

1.4. Управление конфигурациями артефактов базы данных

Иногда обнаруживается то, что изменение в системе оказалось неудачным, поэтому необходимо выполнить откат этого изменения до предыдущего состояния. Например, предположим, что переименование столбца `Customer.FName` в `Customer.FirstName` может нарушить работу пятидесяти внешних программ, а затраты на обновление этих программ в данный момент слишком велики. Для обеспечения успешного проведения рефакторинга базы данных необходимо поставить указанные ниже объекты под контроль средств управления конфигурациями.

- Сценарии на языке DDL (Data Definition Language — язык определения данных), предназначенные для создания схемы базы данных.
- Сценарии загрузки/извлечения данных.
- Файлы с определениями моделей данных.
- Метаданные объектно-реляционного отображения.
- Справочные данные.
- Определения хранимых процедур и триггеров.
- Определения представлений.
- Ограничения ссылочной целостности.
- Другие объекты базы данных, такие как последовательности, индексы и т.д.
- Тестовые данные.
- Сценарии создания тестовых данных.
- Тестовые сценарии.

1.5. Варианты среды, предназначенные для разработчиков

Специализированная среда разработки представляет собой полностью функциональную среду, позволяющую обеспечить создание, тестирование и (или) эксплуатацию системы. Различные специализированные варианты среды должны быть обособлены по соображениям безопасности, поскольку необходимо предоставить разработчикам возможность работать в собственной специализированной среде без опасений нанести урон усилиям других разработчиков; группе обеспечения качества/тестирования — надежно выполнять свои тесты интеграции системы; а конечным пользователям — эксплуатировать свои приложения, не беспокоясь о том, что разработчики внесут искажения в их исходные данные и (или) нарушат функционирование системы. Логическая организация специализированных вариантов среды показана на рис. 1.5; авторы именуют эту организацию логической, поскольку крупная/сложная среда может состоять из семи или восьми физических специализированных вариантов среды, тогда как небольшая/простая среда может включать только два или три физических специализированных варианта среды.

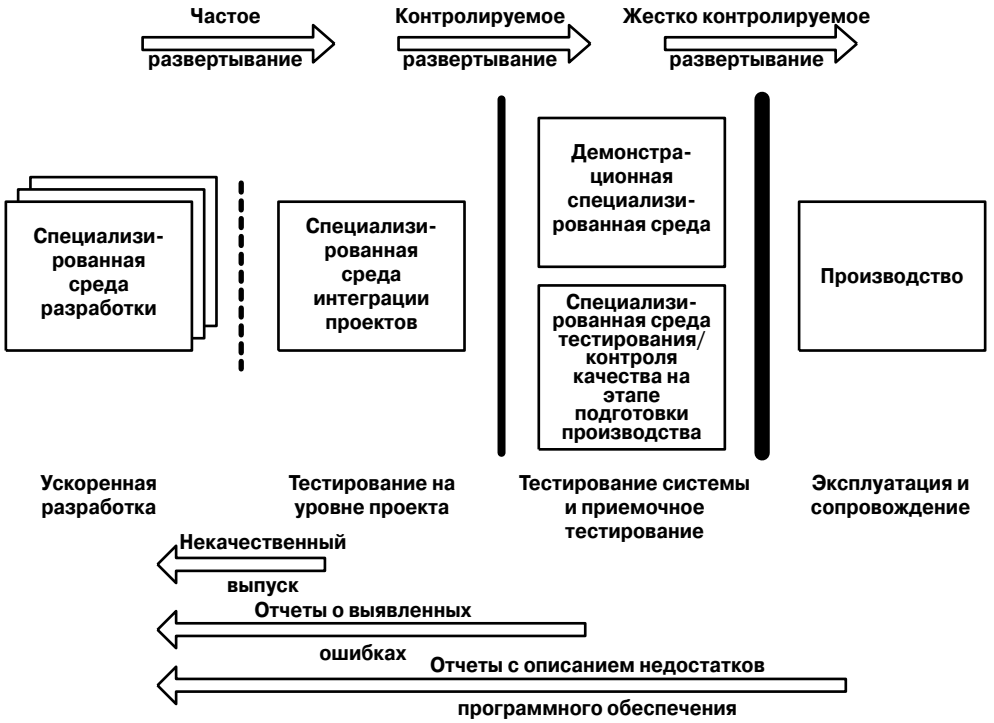


Рис. 1.5. Логические специализированные варианты среды, позволяющие разработчикам работать в безопасности

Чтобы успешно заниматься рефакторингом схемы базы данных, разработчики должны иметь собственные физические специализированные варианты среды для работы, копию исходного кода для доработки и копию базы данных для работы с ней и доработки. При наличии собственной среды разработчики получают возможность без опасений вносить изменения, тестировать их, а затем либо принимать, либо отвергать эти изменения. Полностью убедившись в том, что предлагаемая операция рефакторинга базы данных является осуществимой, разработчики переносят эту операцию в совместно используемую среду проектирования, проводят тестирование в этой среде, затем передают функции по осуществлению операции в систему управления изменениями, после чего ответственность за проведение рефакторинга возлагается на других представителей группы разработчиков. В конечном итоге группа разработчиков передает выполненную ею работу, включая все подготовленные операции рефакторинга базы данных, в демонстрационную среду и (или) в среду тестирования на этапе подготовки к передаче на производство. Такая окончательная передача на производство часто происходит после завершения цикла разработки, но ее осуществление может быть предусмотрено более или менее часто в зависимости от конкретной среды. (Чем чаще вносят изменения в производственную систему, тем выше шансы на получение ценной обратной связи.) Наконец, после того как предлагаемая система пройдет приемку и системное тестирование, происходит ее развертывание на производстве. Этот процесс продвижения/развертывания более подробно рассматривается в главе 4 “Развертывание на производстве”.

1.6. Недостатки эволюционных методик разработки баз данных, препятствующие их внедрению

Авторы посчитали бы это упущением, если бы не рассмотрели вопрос о том, какие препятствия обычно возникают при использовании методик, описанных в этой книге. Первым препятствием, наиболее сложным для преодоления, являются сложившиеся традиции. Многие из современных специалистов в области баз данных начинали свою карьеру в 1970-х–начале 1980-х годов, когда широко применялся подход к разработке по принципу "написания кода, а затем его исправления". Сообщество разработчиков в области информационной технологии признало, что применение этого подхода ведет к созданию кода низкого качества, сложного в сопровождении; в связи с этим на вооружение были приняты трудоемкие методики структурированной разработки, которыми до сих пор еще пользуются многие. Наблюдая за этими экспериментами, большинство специалистов в области баз данных пришли к выводу, что эволюционные методики, появившиеся в результате революции в объектной технологии в 1990-х годах, представляют собой лишь новую интерпретацию подходов по принципу "разработки кода и его исправления", которые доминировали в 1970-х годах; ради справедливости следует отметить, что многие программисты-практики, использующие объектные технологии, действительно организуют свою работу по указанному принципу. Таким образом, многие специалисты в области баз данных поставили знак равенства между эволюционными подходами и низким качеством, но, как показали представители сообщества разработчиков, которые используют адаптивные методологии, фактически дело обстоит иначе. Однако конечным результатом всеобщей приверженности устаревшим подходом стало то, что, по-видимому, основная часть литературы, посвященной тематике применения данных, не выходит за рамки традиционных, последовательных процессов разработки, относящихся к прошлому, и поэтому адаптивные подходы главным образом игнорируются. Итак, сообществом специалистов в области баз данных потеряно много времени, и теперь необходимо наверстать упущенное.

Вторым препятствием является нехватка инструментальных средств, но этот недостаток быстро компенсируется в результате деятельности в области разработки программ с открытым исходным кодом (по крайней мере, в сообществе специалистов, работающих на языке Java). Безусловно, большие усилия были затрачены на разработку инструментальных средств объектно-реляционного отображения и созданы некоторые инструментальные средства тестирования баз данных, но все еще необходимо выполнить большой объем работы. Следует отметить, что поставщикам инструментальных средств программирования потребовалось несколько лет для реализации функциональных средств рефакторинга в своем инструментарии (в действительности в наши дни нелегко найти современную интегрированную среду разработки, которая не предоставляет такие функции), а это означает, что поставщикам инструментальных средств для баз данных также потребуется несколько лет для достижения аналогичной цели. Существует явная потребность в создании удобных и гибких инструментальных средств, которые обеспечивали бы эволюционную разработку схем баз данных; практика показывает, что сообщество разработчиков программ с открытым исходным кодом начинает восполнять такую нехватку, и, по мнению авторов, в конечном итоге этим начнут заниматься и коммерческие поставщики инструментальных средств.

1.7. Резюме

Стандартом современной разработки программного обеспечения фактически стали эволюционные подходы к разработке, которые по своему характеру являются итеративными и инкрементными. После того как определенная группа проектировщиков принимает решение придерживаться в разработке эволюционного подхода, каждый представитель этой группы, включая профессиональных специалистов в области обработки данных, обязан организовывать свою работу на основе принципов эволюционной разработки. К счастью, существуют и такие эволюционные методики, которые позволяют специалистам в области обработки данных принять на вооружение эволюционный подход. К этим методам относятся: рефакторинг базы данных; эволюционное моделирование данных; регрессионное тестирование базы данных; управление конфигурациями артефактов, относящихся к области обработки данных; а также создание отдельных специализированных вариантов среды разработки.

Глава 2

Операции рефакторинга базы данных

Как только усовершенствование проекта заканчивается, проект становится устаревшим.

Фред Брукс

В настоящей главе приведен краткий обзор фундаментальных концепций, лежащих в основе рефакторинга базы данных, и описано, что представляет собой рефакторинг, как он вписывается в общую деятельность по разработке и почему так часто возникают затруднения при его осуществлении. В следующих главах будет подробно показано, как фактически осуществляется процесс рефакторинга схемы базы данных.

2.1. Рефакторинг кода

В своей книге *Refactoring* Мартин Фаулер [**17**] описывает методику программирования, называемую рефакторингом, которая представляет собой регламентированный способ реструктуризации кода с применением небольших шагов. Рефакторинг обеспечивает постепенное развитие кода во времени, в результате чего реализуется эволюционный (т.е. итеративный и инкрементный) подход к программированию. Характерной особенностью рефакторинга является то, что он сохраняет функциональную семантику кода. В результате проведения рефакторинга не происходит ни расширение, ни сужение функциональных возможностей кода. Рефакторинг просто способствует улучшению проекта кода. Например, на рис. 2.1 показано, как применяется операция рефакторинга "Перенос метода на нижний уровень" для переноса метода `calculateTotal()` из класса `Offering` в его подкласс `Invoice`. На первый взгляд это изменение выглядит простым, но следует учитывать, что может также потребоваться внести изменения в код, в котором вызывается указанный метод, для работы с объектами `Invoice`, а не с объектами `Offering`. Но, после того как все необходимые изменения будут внесены, можно утверждать, что код действительно был подвергнут рефакторингу, поскольку он снова работает как и прежде.

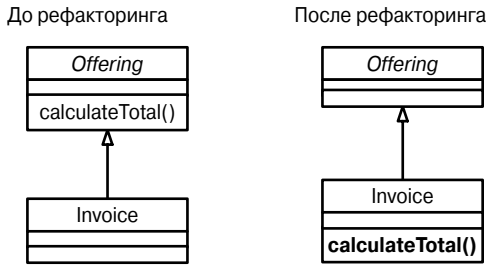


Рис. 2.1. Перенос метода из класса в подкласс

Очевидно, что для проведения рефакторинга кода должен быть предусмотрен систематический подход, в том числе удобные инструментальные средства и методики. В настоящее время рефакторинг кода до некоторой степени поддерживают большинство современных вариантов интегрированной среды разработки (Integrated Development Environment — IDE), а это во многом нас обнадеживает. Но, для того чтобы обеспечить проведение операций рефакторинга на практике, необходимо также разработать современный набор для регрессионного тестирования, позволяющий убедиться в том, что код по-прежнему успешно работает, поскольку нельзя слепо доверять методам рефакторинга кода, не имея возможности полностью убедиться в том, что в результате их применения работа кода не была нарушена.

Многие разработчики, организующие свою работу на принципах адаптивного программирования, и, в частности, специалисты по экстремальному программированию (Extreme Programmer — XPer), полагают, что рефакторинг является ведущим подходом к разработке. На практике можно столь же часто встретить примеры применения рефакторинга небольших фрагментов кода, как и введения операторов `if` или циклов. Рефакторинг кода должен осуществляться до полного исчерпания его возможностей, поскольку наибольшая производительность может быть достигнута только в условиях работы с исходным кодом максимально высокого качества. При возникновении необходимости добавить к коду новые возможности первым делом следует найти ответ на вопрос, действительно ли этот код имеет самый лучший возможный проект, позволяющий успешно реализовать требуемые средства. Если ответ на этот вопрос является положительным, можно сразу же приступать к добавлению новых функциональных средств. Если же ответ отрицателен, то код вначале должен быть подвергнут рефакторингу, для того чтобы он имел самый лучший возможный проект, и только после этого можно приступать к добавлению новых функций. На первый взгляд создается впечатление, что применение указанного подхода приводит к значительному увеличению объема работы, но практика показывает, что если доработка начинается с высококачественного исходного кода, после чего постоянно осуществляется рефакторинг этого кода для поддержки его в том же состоянии, то все новые замыслы реализуются чрезвычайно успешно.

2.2. Рефакторинг баз данных

Рефакторингом базы данных [**4**] называется простое изменение в схеме базы данных, способствующее улучшению проекта базы данных и вместе с тем обеспечивающее сохранение функциональной и информационной семантики базы данных; иными словами, проведение операций рефакторинга не должно приводить к добавлению новых функциональных средств или нарушению работы существующих, а также не должно иметь своим следствием добавление новых данных или изменение смысла существующих данных. Авторы считают, что схема базы данных включает и структурные аспекты, такие как определения таблиц и представлений, и функциональные аспекты, такие как хранимые процедуры и триггеры. Исходя из этой точки зрения, авторы используют термин *рефакторинг кода* для обозначения операций, которые принято называть *операциями рефакторинга*, описанных Мартином Фаулером, а термин *рефакторинг базы данных* — для обозначения операций рефакторинга схемы базы данных. Процесс рефакторинга базы данных, подробно описанный в главе 3 "Процесс рефакторинга базы данных", представляет собой действия по внесению указанных простых изменений в схему базы данных.

Операции рефакторинга базы данных концептуально являются более сложными, чем операции рефакторинга кода, поскольку при проведении операций рефакторинга кода необходимо заботиться лишь о сохранении функциональной семантики, а при осуществлении операций рефакторинга базы данных необходимо также обеспечить сохранение информационной семантики. Еще более важным фактором является то, что усложнение операций рефакторинга базы данных может быть обусловлено наличием большого количества связей, поддерживаемых архитектурой базы данных (рис. 2.2). *Связность* — это мера зависимости между двумя объектами; чем более тесно связаны два предмета, тем выше вероятность того, что изменение в одном из них потребует внесения изменений в другой. Простейшая ситуация возникает при использовании архитектуры базы данных, предназначенной для одного приложения; в этом случае с базой данных взаимодействует только одно известное приложение, а это позволяет одновременно подвергнуть рефакторингу и приложение, и базу данных, после чего сразу же выполнить развертывание всех изменений. Такие ситуации действительно встречаются на практике, и применяемые при этом средства доступа к данным и обработки данных часто именуются *автономными приложениями*, или *неразветвленными системами* (stovepipe system). Вторая архитектура, приведенная в качестве примера на рис. 2.2, является намного более сложной, поскольку характеризуется наличием большого количества внешних программ, взаимодействующих с базой данных, причем некоторые из этих программ выходят за пределы контроля тех, кто эксплуатирует базы данных. В этой ситуации нельзя рассчитывать на то, что может быть обеспечено развертывание изменений одновременно во всех внешних программах, поэтому должен быть предусмотрен переходный период (называемый также *периодом поддержки устаревшего кода*), в течение которого необходимо поддерживать параллельно и старую, и новую схему. Дополнительная информация на эту тему приведена ниже.

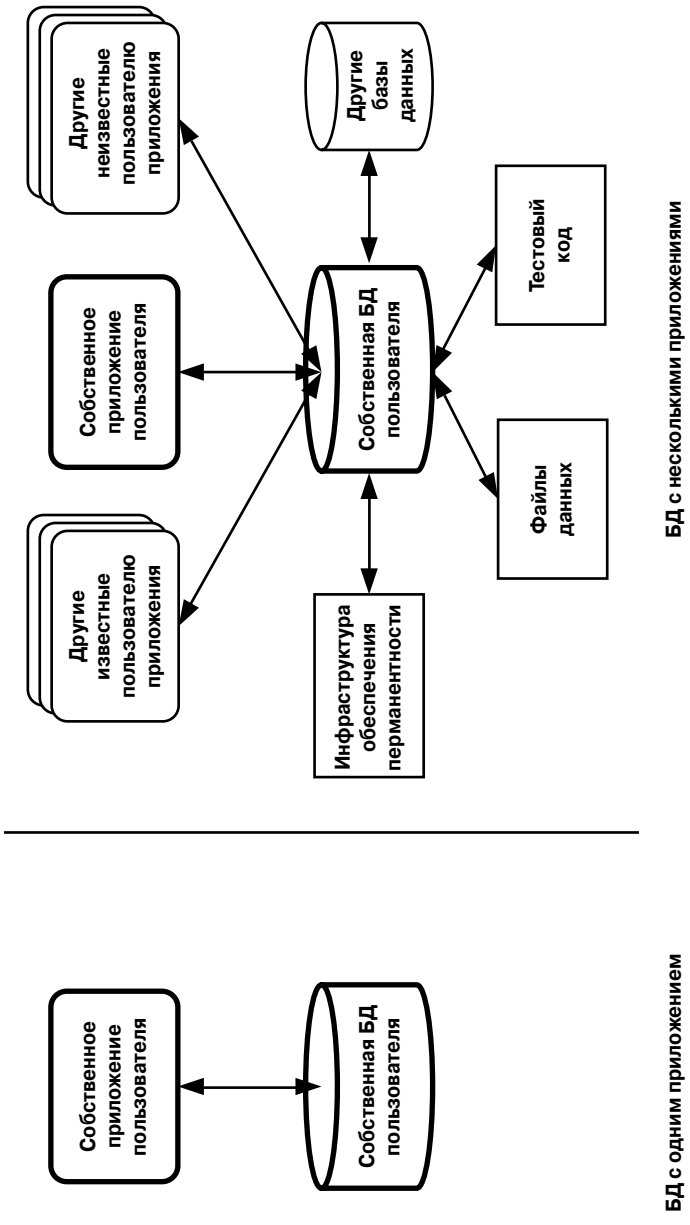


Рис. 2.2. Две категории архитектур базы данных

Безусловно, в этой книге определенное внимание будет уделено описанию среды с одним приложением, но в ней в основном рассматривается среда с многочисленными приложениями, в которой база данных в настоящее время применяется на производстве и доступ к ней осуществляется с помощью многих других внешних программ, над которыми организация, эксплуатирующая базу данных, имеет лишь небольшой контроль или вообще его не имеет. Но эта ситуация не является безвыходной. В главе 3 описаны стратегии, позволяющие успешно работать и в подобных условиях.

Рассмотрим краткий пример, который позволяет понять, при каких обстоятельствах обычно осуществляется рефакторинг базы данных. В этом примере речь идет о том, что в течение нескольких недель происходит эксплуатация банковского приложения, после чего обнаруживается нечто странное по отношению к таблицам `Customer` и `Account` (рис. 2.3). В частности, возникает вопрос, действительно ли имеет смысл, чтобы столбец `Balance` был частью таблицы `Customer`. Ответ на этот вопрос оказался отрицательным, поэтому было решено применить операцию рефакторинга "Перемещение столбца" (с. 139) для усовершенствования проекта базы данных.

2.2.1. Среда базы данных с одним приложением

Начнем с изучения примера перемещения столбца из одной таблицы в другую в среде базы данных с одним приложением. Это — наиболее простая ситуация, с которой когда-либо приходится сталкиваться, поскольку организация, эксплуатирующая базу данных, имеет полный контроль и над схемой базы данных, и над исходным кодом приложения, которое получает доступ к базе данных. Из этого следует, что существует возможность одновременно подвергнуть рефакторингу и схему базы данных, и прикладной код; иными словами, не требуется поддерживать параллельно исходную и новую схемы базы данных, поскольку доступ к базе данных получает только одно приложение.

Авторы предлагают, чтобы в этом сценарии две группы специалистов (или два человека) работали одновременно как один коллектив; специалисты одной группы должны обладать навыками прикладного программирования, а специалисты другой — навыками разработки базы данных, а в идеальном случае специалисты обеих групп должны обладать и теми и другими навыками. Специалисты обеих групп должны начать работу с определения того, должна ли схема базы данных быть подвергнута рефакторингу. Ведь программисты, возможно, ошибаются по поводу того, что требуется обеспечить развитие схемы, а также не знают, как лучше всего осуществить требуемую операцию рефакторинга. Операция рефакторинга вначале разрабатывается и проверяется в специализированной среде разработчика. После завершения этого этапа изменения переносятся в среду интеграции проектов, после чего происходит доработка, тестирование и исправление системы по мере необходимости.

В ходе применения операции рефакторинга "Перемещение столбца" (с. 139) в специализированной среде разработки специалисты обеих групп прежде всего выполняют все тесты, чтобы проверить, проходит ли их система. После этого разработчики пишут еще один тест, поскольку должны руководствоваться подходом к разработке на основе тестирования (Test-Driven Development — TDD).

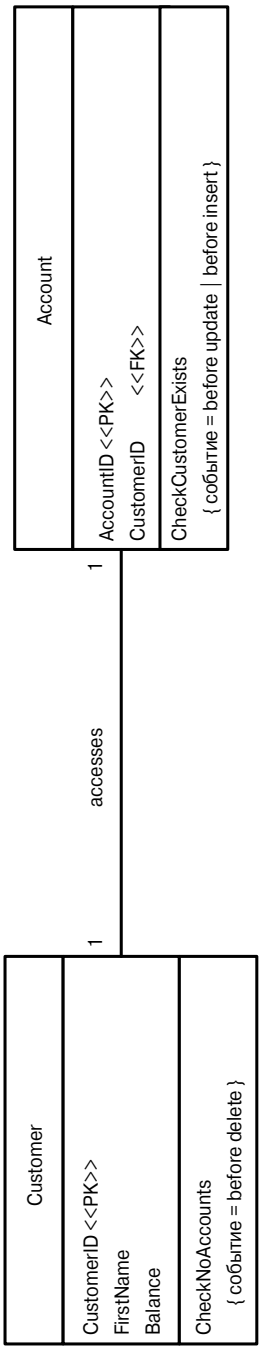


Рис. 2.3. Фрагмент исходной схемы базы данных, относящийся к таблицам `Customer` *и* `Account`

По всей видимости, тест должен предусматривать получение доступа к какому-либо значению в столбце Account.Balance. Перемещение этого столбца еще не выполнено, поэтому тест должен окончиться неудачей; проведя прогон новых тестов и убедившись в том, что они действительно не проходят успешно, разработчики добавляют столбец Account.Balance (рис. 2.4). После этого разработчики повторно запускают тесты на выполнение, чтобы убедиться в том, что теперь эти тесты проходят. Затем разработчики подвергают рефакторингу существующие тесты, чтобы иметь возможность проверить, что функции проверки остатков на счетах клиентов работают должным образом со столбцом Account.Balance, а не со столбцом Customer.Balance.

Убедившись в том, что эти тесты терпят неудачу, разработчики приступают к переопределению функциональных средств проверки остатков на счетах клиентов, чтобы в них использовался столбец Account.Balance. Кроме того, разработчики вносят аналогичные изменения в прочий код, относящийся к набору тестов и к приложению, такой как программные средства списания со счета, в котором в настоящее время предусмотрено использование столбца Customer.Balance.

После того как приложение снова начинает успешно функционировать, разработчики выполняют в целях обеспечения сохранности данных резервное копирование данных столбца Customer.Balance, а затем копируют данные из строк столбца Customer.Balance в соответствующие строки столбца Account.Balance. После этого они повторно запускают свои тесты, чтобы убедиться в том, что перемещение данных было выполнено без потери функциональных возможностей. Последним шагом, завершающим рассматриваемые изменения в схеме, становится удаление столбца Customer.Balance, а затем повторное выполнение всех тестов и устранение по мере необходимости всех обнаруженных нарушений в работе. После окончания описанной последовательности действий разработчики распространяют осуществляемые ими изменения на среду интеграции проектов, как было описано выше.

2.2.2. Среда базы данных с несколькими приложениями

Эта ситуация является более затруднительной, поскольку развертывание новых выпусков отдельных приложений должно происходить в разное время в течение следующих полутора лет. Для реализации этой операции рефакторинга базы данных необходимо выполнить такую же работу, как и в среде базы данных с одним приложением, за исключением того, что столбец Customer.Balance не подлежит немедленному удалению. Вместо этого в течение "переходного периода", составляющего по меньшей мере полтора года, оба столбца должны эксплуатироваться параллельно, для того чтобы группы разработчиков имели достаточно времени для обновления и повторного развертывания всех своих приложений. Эта часть схемы базы данных, применяемой в течение переходного периода, показана на рис. 2.5. Обратите внимание на то, как используются два триггера, SynchronizeCustomerBalance и SynchronizeAccountBalance, эксплуатируемые на производстве в течение переходного периода, для обеспечения синхронизации двух столбцов.

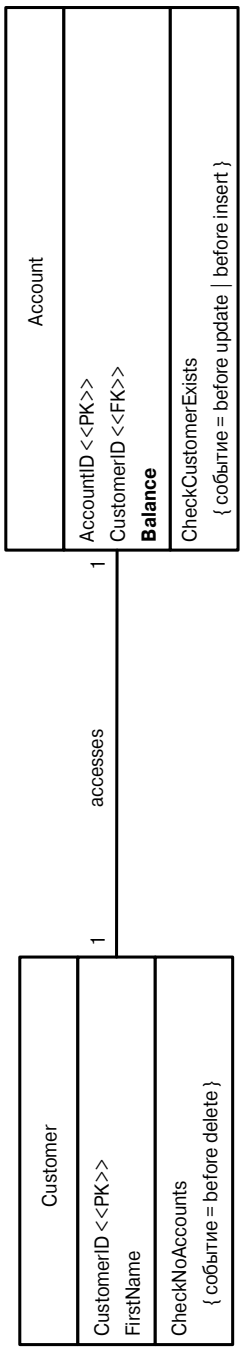


Рис. 2.4. Фрагмент окончательно полученной схемы базы данных, относящийся к таблицам `Customer` *и* `Account`

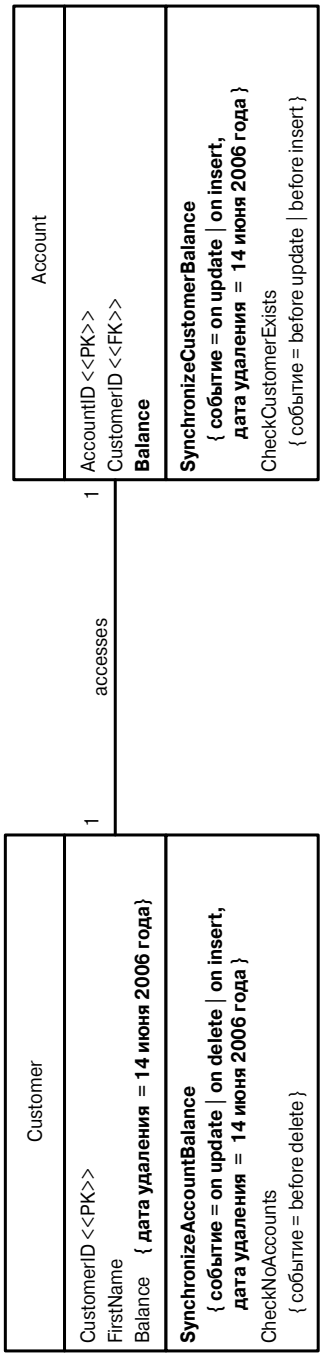


Рис. 2.5. Схема базы данных, применяемая на протяжении переходного периода

Такая значительная продолжительность переходного периода является вполне оправданной. Дело в том, что над некоторыми приложениями в настоящее время еще никто не работает, тогда как другие приложения создаются в рамках традиционного жизненного цикла разработки и выпуск новых версий происходит примерно один раз в год, а при определении длительности переходного периода должны учитываться требования не только тех групп разработчиков, которые часто подготавливают новые выпуски, но и групп, создающих новые версии гораздо реже. Кроме того, мы не можем руководствоваться предположением, что во всех отдельно взятых приложениях будет предусмотрено обновление обоих столбцов, поэтому для синхронизации значений этих столбцов необходимо предусмотреть какой-то механизм, подобный триггерам. Безусловно, могут быть предложены и другие способы синхронизации, например, основанные на использовании представлений или на проведении синхронизации после каждой операции модификации данных, но, как будет описано в главе 5 "Стратегии рефакторинга баз данных", авторы пришли к выводу, что триггеры выполняют эту работу лучше всего.

По завершении переходного периода осуществляется удаление исходного столбца, а также триггера (триггеров), в результате чего создается окончательная схема базы данных (см. рис. 2.4). Удаление этих объектов должно производиться только после тщательного тестирования, позволяющего убедиться в том, что это действие не приведет к каким-либо нарушениям в работе. После этого операция рефакторинга считается выполненной. В главе 3 реализация данного примера на практике будет рассматриваться более подробно.

2.2.3. Сохранение семантики

Проводя операции рефакторинга схемы базы данных, необходимо стремиться сохранить информационную и функциональную семантику; иными словами, в схему не должно ничего добавляться, а также изыматься. Под информационной семантикой подразумевается смысл информации, находящейся в базе данных, с точки зрения пользователя этой информации. Соблюдение требования по сохранению информационной семантики означает, что при любом изменении значений данных, хранящихся в некотором столбце, клиенты, использующие эту информацию, не должны испытывать каких-либо отрицательных последствий такого изменения; например, если к столбцу с символьными значениями номеров телефонов применяется операция рефакторинга базы данных, подобная операции "Введение общего формата" (с. 210), для преобразования таких данных, как `(416) 555-1234` и `905.555.1212` соответственно, в `4165551234` и `9055551212`, это не должно привести к нарушениям в работе клиентов. Безусловно, новый формат является более удобным по сравнению со старым, а для работы с данными теперь можно использовать более простой код, но с практической точки зрения информационное содержание данных фактически не изменилось. Следует отметить, что даже если будет решено отображать номера телефонов в формате `(XXX) XXX-XXXX`, при заполнении этого формата должны использоваться данные в том виде, в каком они хранятся в базе данных.

При подготовке любой операции рефакторинга базы данных наиболее важным критерием является практическая целесообразность. Когда речь идет о рефакторинге кода, Мартин Фаулер неизменно подчеркивает важность проблемы "наблюдаемого поведения"; под этим подразумевается то, что при проведении многих операций рефакторинга нельзя быть полностью уверенным в том, что семантика не изменится даже в каких-то

мелочах, поэтому нам остается лишь подготовить будущую операцию настолько тщательно, насколько это возможно, написать тесты, которые, по-видимому, будут достаточными, а затем выполнить эти тесты для проверки то, что семантика не изменилась. Опыт авторов показывает, что аналогичные проблемы возникают, когда приходится задумываться о сохранении информационной семантики при проведении операций рефакторинга схемы базы данных, ведь после перехода от формы `(416) 555-1234` к форме `4165551234` фактически может оказаться, что семантика этой информации изменилась для какого-то приложения в том смысле, нюансы которого нам не известны. Например, предположим, что имеется какой-то отчет, который по неизвестным причинам формируется правильно только при том условии, что номера телефонов представлены в строках с данными в формате `(XXX) XXX-XXXX`, и если это условие не соблюдается, попытка формирования отчета приводит к получению непредвиденных результатов. В частности, предположим, что в данном случае вывод номеров телефонов в отчете выполняется в формате `XXXXXXXXXX`, но в результате этого чтение отчета становится более затруднительным, даже несмотря на то, что с точки зрения практики происходит вывод той же информации, что и прежде. После того как эта проблема будет в конечном итоге обнаружена, может потребоваться обновление отчета на основе нового формата.

Аналогичным образом, что касается функциональной семантики, цель ее сохранения состоит в том, чтобы все функциональные средства, применяемые по принципу черного ящика, остались неизменными; это означает, что весь исходный код, используемый для работы с изменившимися частями схемы базы данных, должен быть переработан в целях предоставления тех же функциональных возможностей, что и прежде. Например, после проведения операции рефакторинга "Введение вычислительного метода" (с. 268) может потребоваться внести изменения во все прочие существующие хранимые процедуры, чтобы обеспечить в них вызов вновь введенного метода, а не реализовывать повторно одни и те же алгоритмы для проведения соответствующих вычислений. В данном случае в базе данных по-прежнему остаются реализованными те же алгоритмы, но программные средства выполнения вычислений по этим алгоритмам концентрируются в одном месте.

При этом необходимо учитывать, что операции рефакторинга базы данных представляют собой подмножество преобразований базы данных. *Преобразование базы данных* может предусматривать или не предусматривать изменение семантики, а проведение любой операции рефакторинга базы данных не должно повлечь за собой изменение семантики. В главе 11 "Преобразования" приведено описание некоторых распространенных преобразований базы данных; это сделано не только потому, что важно иметь о них представление, но и потому, что преобразования часто становятся одним из этапов операции рефакторинга базы данных. Например, применяя упомянутую выше операцию "Перемещение столбца" для перемещения столбца `Balance` из таблицы `Customer` в таблицу `Account`, приходится осуществлять на одном из этапов преобразование "Введение нового столбца" (с. 321).

На первый взгляд создается впечатление, что операция "Введение нового столбца" полностью соответствует определению операции рефакторинга, ведь добавление пустого столбца к таблице не изменяет семантику этой таблицы до тех пор, пока этот столбец не используется в каких-либо новых функциональных средствах. Тем не менее авторы рассматривают эту операцию как преобразование (а не как рефакторинг), поскольку непродуманное применение этой операции может привести к изменению функционирования приложения. Например, если новый стобец будет введен в середину таблицы, то работа любых программных средств, в которых используется позиционный доступ (например, работа ко-

да, который ссылается на столбец 17, а не на имя столбца), нарушится. Кроме того, перестанет работать код COBOL, привязанный к таблице базы данных DB2, если не будет выполнена его повторная привязка к новой схеме, даже если столбец добавлен в конце таблицы. Таким образом, прежде всего необходимо руководствоваться соображениями практической целесообразности. Дело в том, что последствия применения операции "Введение нового столбца" останутся одинаковыми, независимо от того, назовем ли мы ее операцией рефакторинга или как-то иначе. Но, по крайней мере, следует всегда учитывать, для чего предназначены те или другие операции.

Необходимость применения операций рефакторинга

Авторам часто приходится слышать утверждения специалистов в области обработки данных, что лучше всего сразу учесть в модели все требования к схеме базы данных и тогда не потребуется проведение рефакторинга этой схемы. Разумеется, такая точка зрения имеет право на существование, и авторы действительно были свидетелями того, как в некоторых ситуациях указанный подход оправдывается, но опыт, приобретенный всем сообществом специалистов в области информационной технологии за последние три десятилетия, показывает, что на практике невозможно все продумать заранее. В этом традиционном подходе к моделированию данных не учтены возможности таких современных эволюционных методов, как RUP и XP, а также игнорируется тот факт, что клиенты деловых предприятий все чаще выдвигают требования по внедрению новых средств и внесению изменений в существующие функциональные возможности. Применявшиеся ранее принципы разработки, согласно которым все должно быть сделано раз и навсегда, больше никого не устраивают.

Как было указано в главе 1 "Эволюционная разработка баз данных", авторы предлагают руководствоваться подходом AMDD (Agile Model-Driven Development — адаптивная разработка на основе модели), который предусматривает осуществление определенного моделирования высокого уровня для определения общего "ландшафта" своей системы, а затем оперативно проводить доработку в модели конкретных деталей на основе текущих потребностей (Just-In-Time — JIT). Это позволяет воспользоваться преимуществами моделирования и вместе с тем избавиться от лишних издержек, связанных с чрезмерно тщательным моделированием, составлением ненужной документации и укреплением бюрократического аппарата, без которого невозможно поддерживать слишком большое количество артефактов в актуальном состоянии и обеспечивать их синхронизацию друг с другом. Прикладной код и схема базы данных должны развиваться по мере углубления понимания предметной области, а поддержка высокого качества того и другого должна осуществляться с помощью операций рефакторинга.

2.3. Категории операций рефакторинга базы данных

Следует отметить, что авторы различают шесть разных категорий операций рефакторинга базы данных (табл. 2.1). Такая стратегия распределения по категориям была введена для улучшения организации изложения материала в настоящей книге, и мы надеемся, что она поможет лучше организовать разработку перспективных инструментальных средств рефакторинга баз данных. Но предложенная нами стратегия категоризации далека от совершенства; например, операция рефакторинга "Замена метода (методов) пред-

ставлением" (с. 287) может быть отнесена и к категории рефакторинга архитектуры, и к категории рефакторинга методов. (Мы поместили ее в категорию рефакторинга архитектуры.)

Таблица 2.1. Категории операций рефакторинга базы данных

Категория операций рефакторинга базы данных	Описание	Пример (примеры)
Операции рефакторинга структуры (глава 6)	Изменения в определении одной или нескольких таблиц или представлений	Перемещение столбца из одной таблицы в другую или разбиение многоцелевого столбца на несколько отдельных столбцов, каждый из которых выполняет отдельное назначение
Операции рефакторинга качества данных (глава 7)	Изменения, которые позволяют повысить качество информации, хранящейся в базе данных	Создание столбца, не допускающего применения NULL-значений, для обеспечения того, чтобы в нем всегда находились содержательные значения, или применение к столбцу общего формата для обеспечения согласованности
Операции рефакторинга ссылочной целостности (глава 8)	Изменения, которые гарантируют, что указанная в ссылке строка существует в другой таблице, и (или) позволяют обес печить удаление должным образом строк, которые становятся больше не нужными	Добавление триггера, позволяющего реализовать правило каскадного удаления, охватывающее две сущности, для замены кода, который прежде был реализован вне базы данных
Операции рефакторинга архитектуры (глава 9)	Изменения, способствующие в целом улучшению взаимодействия внешних программ с базой данных	Замена существующей процедуры на языке Java, код которой находится в общей библиотеке кода, хранимой процедурой, находящейся в базе данных. После того как код становится реализованным в виде хранимой процедуры, появляется возможность использования этого кода в приложениях, отличных от приложений на языке Java
Операции рефакторинга методов (глава 10)	Внесение изменений в метод (хранимую процедуру, хранимую функцию или триггер), способствующих улучшению его качества. К методам базы данных применимы многие операции рефакторинга кода	Переименование хранимой процедуры в целях упрощения понимания ее назначения
Преобразования, отличные от операций рефакторинга (глава 11)	Изменения в схеме базы данных, которые приводят к изменению ее семантики	Добавление нового столбца в существующую таблицу

2.4. Признаки нарушений в работе базы данных

Фаулер [**17**] ввел понятие “недостатка кода”, обозначив так общую категорию нарушений в работе кода, которые указывают на необходимость проведения его рефакторинга. К общим недостаткам кода относятся операторы `switch`, слишком крупные методы, дублирующийся код, а также зависимость функций от среды. По аналогии с этим могут быть определены общие недостатки базы данных, которые указывают на потенциальную необходимость проведения ее рефакторинга [**4**]. К этим недостаткам относятся следующие.

- **Многоцелевые столбцы.** Если столбец используется в нескольких целях, то велика вероятность, что существует дополнительный код, который служит для обеспечения использования исходных данных “по назначению”, часто путем проверки значений в одном или нескольких других столбцах. В качестве примера можно привести столбец, в котором хранятся даты рождения клиентов и даты приема на работу сотрудников. Еще худшим последствием применения многоцелевого столбца является то, что он ограничивает возможности используемых функциональных средств; в частности, в рассматриваемом примере не учтено то, что когда-либо может потребоваться хранить информацию о датах рождения сотрудников.
- **Многоцелевые таблицы.** Аналогичным образом, если какая-то таблица используется для хранения данных о сущностях нескольких разных типов, то ее появление в базе данных по всей вероятности является следствием упущения в проекте. В качестве примера можно назвать общую таблицу `Customer`, которая используется для хранения информации и о физических лицах, и о корпорациях. Недостатком подобного подхода является то, что для представления информации о физических лицах и корпорациях используются разные структуры данных, например, применительно к физическим лицам необходимо хранить сведения о фамилии, имени и отчестве, а для корпораций достаточно указать юридическое название. В подобной общей таблице `Customer` неизбежно появляются столбцы с NULL-значениями, относящимися к одним типам клиентов, но не к другим.
- **Избыточные данные.** Наличие избыточных данных — это серьезная проблема в повседневно эксплуатируемых базах данных, поскольку при хранении одних и тех же данных в нескольких местах возникает вероятность нарушения совместимости. Например, во многих организациях обнаруживается, что информация о клиентах хранится в нескольких разных местах. Из-за этого многие компании фактически не имеют возможности составить точный список, позволяющий узнать, кто действительно является их клиентами. Проблема состоит в том, что, по данным одной таблицы, клиент `John Smith` проживает по адресу `123 Main Street`, а в другой таблице указано, что он проживает по адресу `456 Elm Street`. В подобных случаях фактически ситуация такова, что данные относятся к одному и тому же лицу, некогда проживавшему по адресу `123 Main Street`, но сменившему свой адрес в прошлом году; к сожалению, `John Smith` не отправил в компанию столько заявлений об изменении своего адреса, сколько в ней имеется приложений, содержащих эти сведения.

- **Таблицы со слишком большим количеством столбцов.** Если в таблице имеется много столбцов, это можно рассматривать как признак отсутствия слитности в структуре таблицы; по-видимому, в этой таблице представлены данные, относящиеся к нескольким разным сущностям. Предположим, что в таблице `Customer` находятся столбцы, предназначенные для хранения трех разных адресов (адреса поставки, адреса выставления счетов и адреса, применяемого в течение сезона), или нескольких номеров телефонов (домашний телефон, рабочий телефон, сотовый телефон и т.д.). По-видимому, должна быть проведена нормализация этой структуры путем добавления таблиц `PhoneNumber` и `Address`.
- **Таблицы со слишком большим количеством строк.** Наличие крупных таблиц служит признаком проблем производительности. Например, при поиске в таблице с миллионами строк нельзя обойтись без больших затрат времени. В связи с этим может потребоваться разбить таблицу по вертикали, переместив некоторые столбцы в другую таблицу, или разбить ее по горизонтали, переместив в другую таблицу некоторые строки. Обе эти стратегии способствуют сокращению размеров таблицы, что может привести к повышению производительности.
- **Многозначные столбцы.** Многозначными называются столбцы, в которых в различных позициях представлено несколько разных фрагментов информации. Например, как многозначный рассматривается столбец с идентификатором клиента, в котором первые четыре цифры идентификатора клиента обозначают головное отделение компании этого клиента, поскольку для выявления дополнительной информации приходится выполнять синтаксический анализ значений из этого столбца (допустим, чтобы узнать идентификатор головного отделения). В качестве еще одного примера можно назвать текстовый столбец, используемый для хранения структур данных XML; очевидно, что структуры данных XML могут быть подвергнуты синтаксическому анализу для выявления представленных в них фрагментов с данными. На практике часто обнаруживается необходимость реорганизации многозначных столбцов для разбиения содержащихся в них данных на отдельные поля с данными, чтобы можно было проще проводить обработку этих полей в виде отдельных элементов.
- **Наличие нереализованных изменений.** Если изменения в схеме базы данных давно не проводились, по той причине, что могут возникнуть какие-либо нарушения в работе, например, пятидесяти приложений, которые получают к ней доступ, это может служить наглядным признаком того, что схему базы данных необходимо подвергнуть рефакторингу. Подобные опасения перед возможными нарушениями в работе явно свидетельствуют о неизменном возрастании риска полного отказа системы, а такая ситуация со временем только ухудшается.

Важно понять, что наличие какого-то недостатка отнюдь не означает, что этот объект нельзя использовать; в частности, лимбургский сыр внешне выглядит просто ужасно, но это не означает, что он непригоден для употребления. Но если испорченным кажется молоко, эту проблему нельзя игнорировать. Обнаружив какие-либо недостатки в схеме базы данных, необходимо их проанализировать, оценить значимость и в случае необходимости подвергнуть схему рефакторингу.

2.5. Перспективы дальнейшего распространения операций рефакторинга

Все современные процессы разработки программного обеспечения, включая унифицированный процесс компании Rational (Rational Unified Process — RUP), экстремальное программирование (Extreme Programming — XP), адаптивный унифицированный процесс (Agile Unified Process — AUP), метод Scrum и адаптивный метод разработки систем (Dynamic System Development Method — DSDM), по своему характеру являются эволюционными. Крэйг Ларман [**25**] подытожил результаты исследований, а также свидетельства ошеломляющей поддержки интеллектульных лидеров, принадлежащих к сообществу специалистов в области информационной технологии, и пришел к выводу, что эволюционные подходы к программированию являются доминирующими. А что касается методов создания приложений, предназначенных для обработки данных, то, к сожалению, они в основном являются по своему характеру последовательными и рассчитаны на привлечение специалистов, выполняющих относительно узкие задачи, такие как логическое или физическое моделирование данных. В этом и заключается причина современного неудовлетворительного состояния дел — специалисты в области программирования и специалисты в области обработки данных должны сотрудничать, но те и другие стремятся организовать свою работу по-разному.

Авторы занимают такую позицию, что специалисты в области обработки данных могут извлечь не меньшую пользу от принятия на вооружение современных эволюционных методов, чем разработчики программного обеспечения, и что операции рефакторинга базы данных относятся к числу наиболее важных навыков, которыми должны овладеть специалисты в области обработки данных. Остается только пожалеть о том, что революционные изменения, связанные с внедрением объектного подхода, происходившие в 1990-х годах, не затронули сообщество специалистов в области обработки данных, а это означает, что ими были упущены возможности овладеть эволюционными методами, которые в настоящее время воспринимаются прикладными программистами как должное. Во многих отношениях сообщество специалистов в области обработки данных не затронули также революционные изменения, связанные с внедрением адаптивного подхода, благодаря которому было обеспечено развитие методов эволюционной разработки еще на один шаг, что позволило организовать чрезвычайно тесное сотрудничество и взаимодействие.

Рефакторинг баз данных — это метод реализации баз данных, точно так же, как рефакторинг кода представляет собой метод реализации приложений. Как правило, схема базы данных подвергается рефакторингу в целях упрощения дальнейших дополнений к этой схеме. Кроме того, на практике часто обнаруживается, что в базу данных необходимо внести новое средство, такое как новый столбец или хранимая процедура, но существующий проект нельзя назвать наиболее подходящим для обеспечения поддержки этого нового средства. В таком случае прежде всего осуществляется рефакторинг схемы базы данных, чтобы упростить введение нового средства, а после успешного проведения операции рефакторинга добавляется нужное средство. Преимуществом этого подхода является то, что он позволяет постепенно, но неизменно повышать качество проекта базы данных. В результате осуществления такого процесса не только достигается упрощение понимания и использования базы данных, но и упрощается ее развитие со временем; иными словами, повышается общая продуктивность разработки.

На рис. 2.6 приведен общий обзор наиболее важных направлений деятельности по разработке, которая осуществляется в современных проектах, требующих применения не только объектных технологий, но и технологий реляционных баз данных. Обратите внимание на то, что на этом рисунке все стрелки являются двунаправленными. Это означает, что переход от одного вида деятельности к другому может в случае необходимости происходить и в том и в другом направлении. Необходимо также отметить, что на этой блок-схеме отсутствует какое-либо обозначение начала или конца; иными словами, налицо явное отличие от традиционного, последовательного процесса.

Очевидно, что рефакторинг баз данных составляет лишь часть эволюционного подхода к разработке баз данных. Необходимо также принять на вооружение эволюционный/адаптивный подход к моделированию данных. Остается также необходимость проводить тестирование схемы базы данных и передавать ее в руки специалистов по управлению конфигурациями. Кроме того, по-прежнему требуется выполнять должным образом настройку базы данных. Но авторы вынуждены оставить эти темы для рассмотрения в других книгах.

2.6. Возможности упрощения операций рефакторинга схемы базы данных

Чем больше степень связности объекта, подвергаемого рефакторингу, тем сложнее становится осуществление рефакторинга. Такое утверждение справедливо по отношению к рефакторингу кода, а также, безусловно, относится и к рефакторингу базы данных. Опыт авторов показывает, что *связность* становится особенно серьезной проблемой, если приходится рассматривать функциональные аспекты (например, код); к сожалению, авторы многих книг по базам данных предпочитают не затрагивать эту тему. Очевидно, что самым легким сценарием является наличие базы данных с одним приложением, поскольку достаточно учесть лишь внутреннюю связность схемы базы данных, а также ее связь с приложением. Если же приходится сталкиваться с архитектурой базы данных с несколькими приложениями (рис. 2.7), то потенциально схема базы данных может быть связана с исходным кодом приложений, инфраструктурами доступа к базе данных и инструментами объектно-реляционного отображения (Object-Relational Mapping — ORM), с другими базами данных (посредством репликации, извлечения/загрузки данных и т.д.), со схемами файлов данных, с тестовым кодом и даже с самой собой.

Одним из эффективных способов уменьшения степени связности, которой характеризуется база данных, является введение дополнительного уровня доступа к базе данных. Этой цели можно добиться, предоставив доступ внешним программам к базе данных через *уровни обеспечения перманентности* (рис. 2.8). Уровень обеспечения перманентности может быть реализован несколькими способами: с помощью объектов доступа к данным (Data Access Object — DAO), в которых реализован необходимый код SQL; с применением инфраструктур; с помощью хранимых процедур или даже на основе Web-служб. Как показано на рис. 2.8, степень связности никогда не удастся свести к нулю, но ее определенно можно уменьшить до какого-то приемлемого уровня.

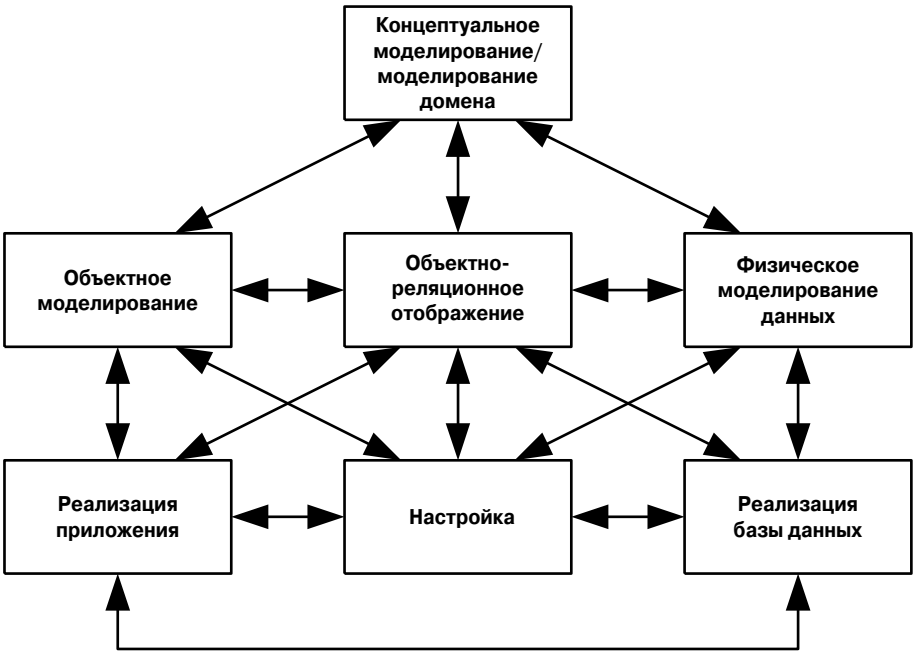


Рис. 2.6. Возможные направления деятельности при разработке эволюционного проекта

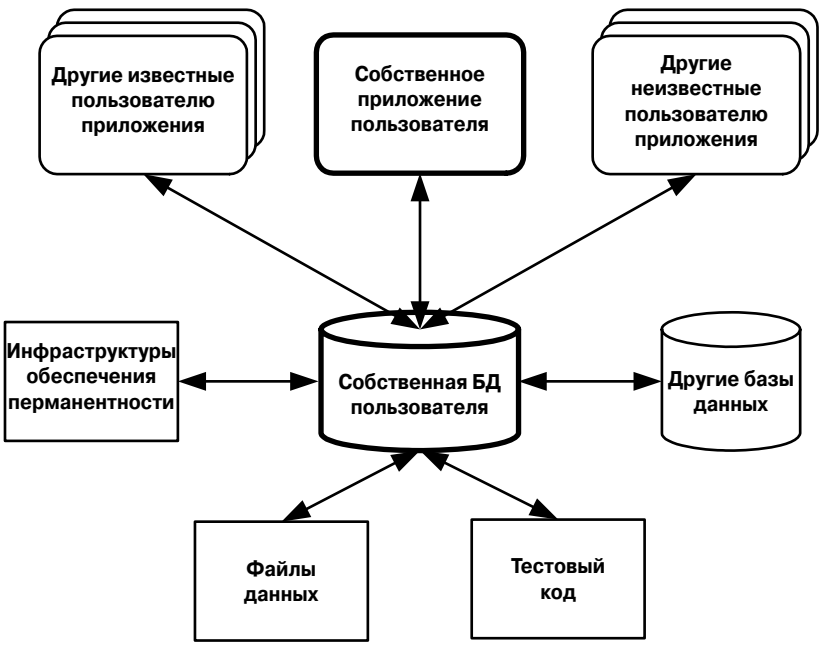


Рис. 2.7. Базы данных, тесно связанные с внешними программами

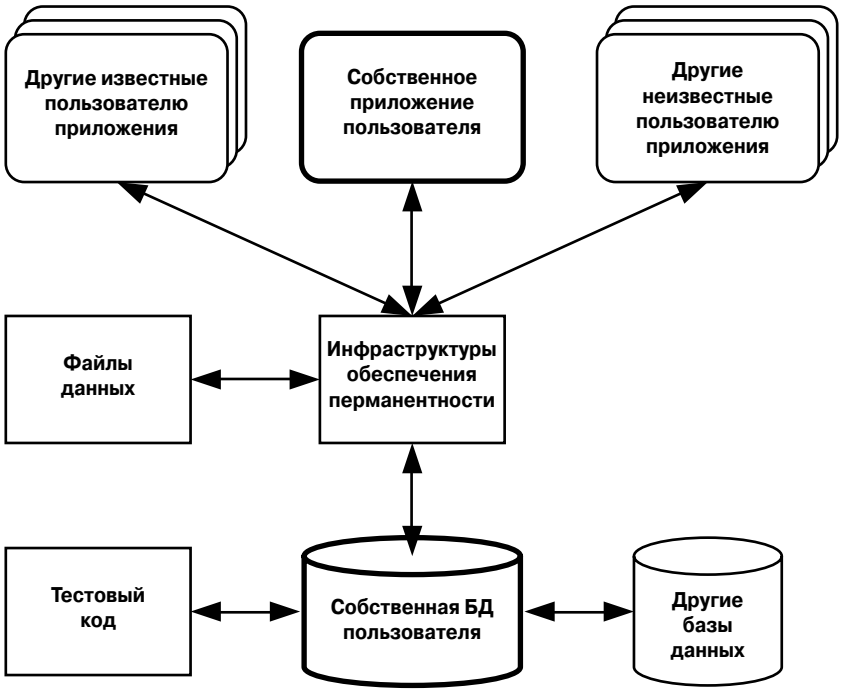


Рис. 2.8. Снижение степени связности путем создания дополнительного уровня доступа

2.7. Резюме

Операция рефакторинга кода — формализованный способ реструктуризации кода с помощью небольших, эволюционных шагов, позволяющий повысить качество проекта кода. В результате проведения операции рефакторинга кода сохраняется функциональная семантика кода; это означает, что выполнение такой операции не приводит ни к добавлению, ни к удалению функциональных средств. По аналогии с этим операция рефакторинга базы данных — это простое изменение в схеме базы данных, которое улучшает ее проект, сохраняя неизменной функциональную и информационную семантику базы данных. Рефакторинг баз данных представляет собой один из основных методов, которые позволяют специалистам в области обработки данных взять на вооружение эволюционный подход к разработке баз данных. Чем больше связность, характерная для базы данных, тем сложнее становится задача применения к ней операций рефакторинга.

Глава 3

Процесс рефакторинга базы данных

Новая научная истина прокладывает дорогу к триумфу не посредством убеждения оппонентов и принуждения их видеть мир в новом свете, но скорее потому, что ее оппоненты рано или поздно умирают и вырастает новое поколение, которое привыкло к ней.

Макс Планк

В настоящей главе показано, как реализовать отдельную операцию рефакторинга в базе данных. В ней рассматривается пример применения операции рефакторинга структуры "Перемещение столбца" (с. 139). На первый взгляд эта операция рефакторинга кажется простой и действительно таковой и является, но в этой главе показано, что подобная операция может стать весьма сложной, если задача состоит в том, чтобы безопасно ее реализовать в производственной среде. На рис. 3.1 схематически показано, как переместить столбец `Customer.Balance` в таблицу `Account`, т.е. выполнить несложное изменение, направленное на улучшение проекта базы данных.

В главе 1 "Эволюционная разработка базы данных" приведен общий обзор понятия логических рабочих специализированных вариантов среды — специализированных вариантов среды разработки, в которых разработчики применяют для работы собственную копию исходного кода и базы данных, среды интеграции проекта, предназначенной для переноса в нее, а затем для проверки предлагаемых изменений членами группы разработчиков, а также среды подготовки производства для проверки системы, интеграции и соответствия требованиям пользователя, и, наконец, производственной среды. Основная работа по подготовке операции рефакторинга базы данных выполняется в специализированной среде разработки; именно в этой среде изменения изучаются, реализуются и тестируются перед передачей в другие варианты среды. Настоящая глава в основном посвящена описанию того, какая работа выполняется в специализированной среде разработки. А в главе 4 "Развертывание на производстве" рассматривается тема передачи в другие варианты среды и окончательного развертывания рассматриваемых операций рефакторинга.

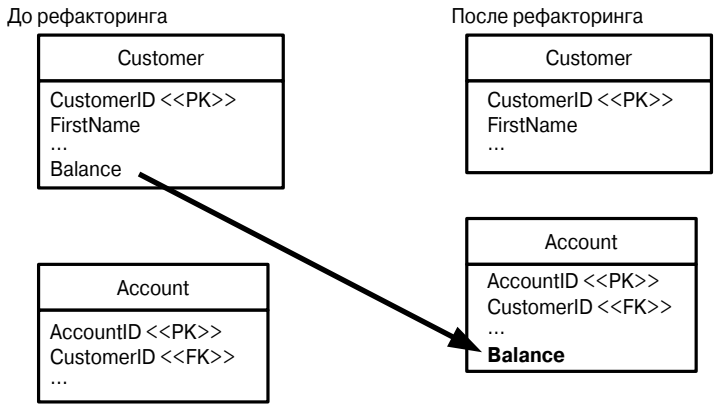


Рис. 3.1. Перемещение столбца `Customer.Balance` *в таблицу* `Account`

Здесь описано, что происходит в специализированной среде разработки, поэтому рассматриваемый процесс применяется и к базе данных с одним приложением, и к варианту среды базы данных с несколькими приложениями. Единственное реальное различие между этими двумя ситуациями состоит в том, что в сценарии с несколькими приложениями требуется более продолжительный переходный период (дополнительная информация на эту тему приведена ниже).

На рис. 3.2 показана блок-схема в обозначениях UML 2, которая представляет собой краткий обзор процесса рефакторинга базы данных. Процесс начинается с того, что разработчик предпринимает попытку реализовать новое требование, направленное на устранение нарушения в работе. Разработчик приходит к выводу, что схема базы данных может потребовать проведения рефакторинга. В этом примере разработчик, которого зовут Эдуард, вводит в свое приложение финансовую транзакцию нового типа и обнаруживает, что столбец `Balance` фактически описывает сущности `Account`, а не сущности `Customer`. Эдуард следует таким общепринятым рекомендациям по адаптивной разработке, как программирование в паре **[34]** и привлечение к моделированию других разработчиков **[3]**, поэтому принимает решение обратиться за помощью к Анжелике, администратору базы данных в группе разработчиков, чтобы она оказала ему содействие в осуществлении операции рефакторинга. Эти разработчики вместе выполняют по итеративному принципу описанные ниже действия.

- Проверка приемлемости рассматриваемой операции рефакторинга базы данных.
- Выбор наиболее подходящей операции рефакторинга базы данных.
- Обозначение как устаревшей исходной схемы базы данных.
- Тестирование до выполнения, во время выполнения и после выполнения операции рефакторинга.
- Внесение изменений в схему базы данных.
- Перемещение исходных данных.
- Внесение изменений во внешнюю программу (программы) доступа.

- Выполнение регрессионных тестов.
- Управление версиями выполняемой работы.
- Публикация сведений о проведенной операции рефакторинга.

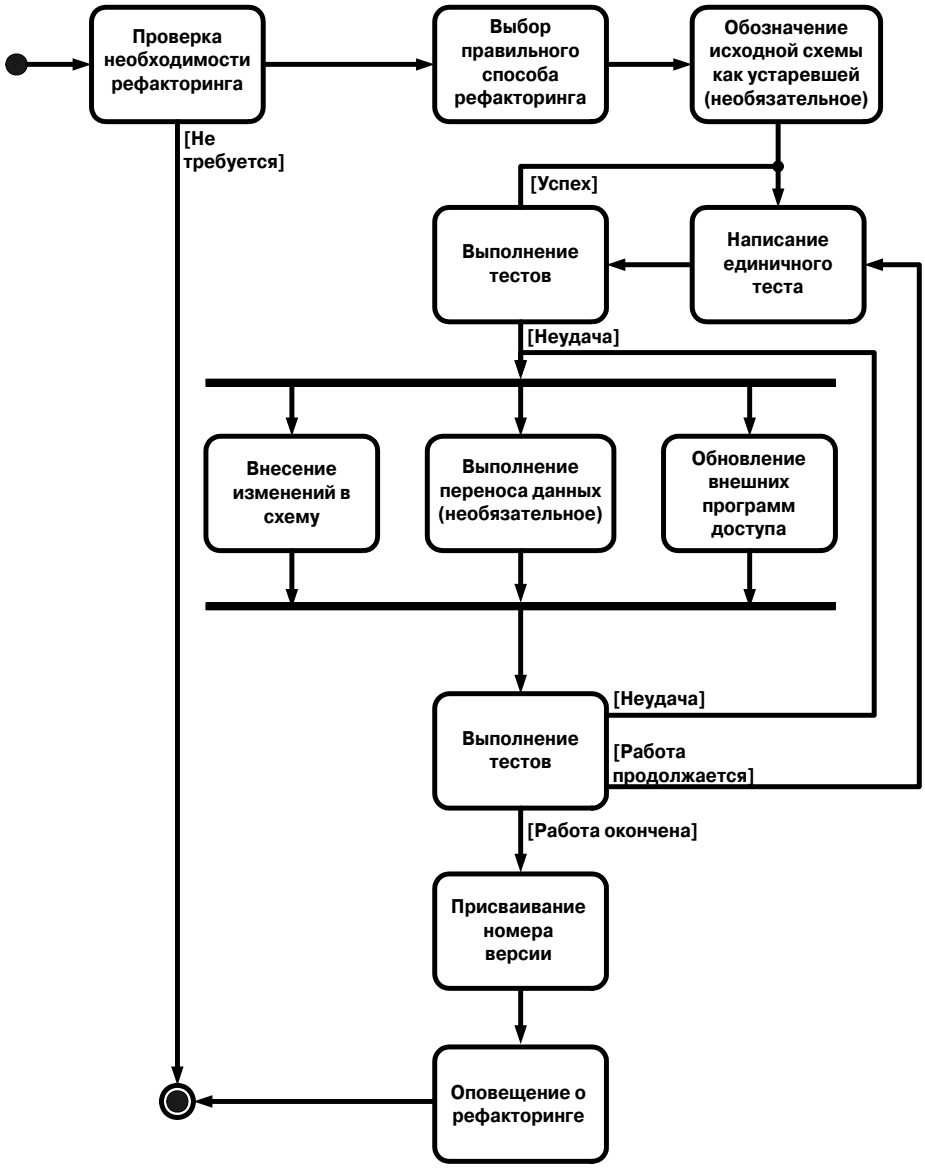


Рис. 3.2. Процесс рефакторинга базы данных

3.1. Проверка приемлемости рассматриваемой операции рефакторинга базы данных

Прежде всего Анжелика определяет, должна ли быть выполнена предложенная операция рефакторинга. Для этого необходимо рассмотреть три описанные ниже проблемы.

1. Имеет ли смысл предлагаемая операция рефакторинга.

   Возможно, существующая структура таблицы является правильной. Разработчики часто не признают правильность существующего проекта базы данных или просто неверно его трактуют. Подобные недоразумения могут привести их к мнению, что проект требует внесения изменений, тогда как в действительности такой необходимости нет. Администратор базы данных должен хорошо знать структуру базы данных, которая используется группой проектировщиков, и других корпоративных баз данных, а также иметь четкое представление о том, к кому обращаться за помощью при решении подобных вопросов. Именно поэтому администратор базы данных имеет решающее слово при определении того, является ли существующая схема самой лучшей. Кроме того, администратор базы данных часто в большей степени понимает общие требования всего предприятия и может дать важную рекомендацию, которая не всегда очевидна для тех, кто рассматривает проблемы управления данными с точки зрения отдельного проекта. Но в рассматриваемом примере создается впечатление, что схема действительно требует внесения изменений.

2. Должны ли быть изменения выполнены немедленно.

   Обычно администратор базы данных примерно знает, чего ожидать, на основе предыдущего опыта сотрудничества с разработчиком приложений. В связи с этим Анжелика проверяет, есть ли у Эдуарда весомые основания для внесения изменений в схему. Не менее важно то, что Эдуард должен объяснить, какие деловые требования поддерживает предлагаемое изменение. Обоснованы ли эти требования? Оправдались ли изменения, которые Эдуард предлагал внести в прошлом? Не приходилось ли Эдуарду отказываться через несколько дней от своего предложения, в связи с чем Анжелика была вынуждена отменять внесенные изменения? В зависимости от такой оценки Анжелика может предложить, чтобы Эдуард подумал над этим изменением еще несколько дней, или может принять решение продолжить работу над реализацией изменения, но выдержать паузу в течение более продолжительного времени, прежде чем приступать к внесению изменения в среду интеграции проектов (подробнее об этом — в главе 4), если, по ее мнению, может возникнуть необходимость в дальнейшем отменить это изменение.

3. Оправдывает ли результат затраченные усилия.

   На следующем этапе Анжелика должна определить, какое влияние в целом окажет выполнение предложенной операции рефакторинга. Чтобы иметь возможность это сделать, Анжелика должна иметь полное представление о том, как внешняя программа (программы) связана с этой частью базы данных. Такие знания Анжелика может приобрести только в течение долгого времени, работая с

архитекторами программного обеспечения предприятия, дежурными администраторами базы данных, разработчиками приложений и другими администраторами баз данных. Если Анжелика не уверена в том, что внесение предлагаемого изменения приведет к положительным результатам, она обязана либо принять решение, руководствуясь своей интуицией, либо посоветовать разработчику немного подождать, пока она не проведет консультации с другими специалистами. Задача Анжелики как администратора базы данных состоит в том, чтобы проводимые ею операции рефакторинга базы данных достигали успеха; если же будет обнаружена необходимость обновить, протестировать и снова развернуть для поддержки операции рефакторинга 50 других приложений, то, возможно, нет смысла продолжать работу в этом направлении. Даже если доступ к базе данных осуществляет только одно приложение, оно может оказаться настолько тесно связанным с той частью схемы, в которой должны быть внесены изменения, что операция рефакторинга базы данных становится нецелесообразной. Но в рассматриваемом примере очевидно, что обнаруженный недостаток проекта является очень серьезным, поэтому Анжелика принимает решение реализовать предложенное изменение, несмотря на то, что могут оказаться затронутыми многие приложения.

Отказ от осуществления крупных изменений

Операции рефакторинга схемы базы данных должны обеспечивать возможность внесения изменений с применением небольших этапов; каждая операция рефакторинга должна выполняться отдельно. Предположим, например, что в процессе эксплуатации обнаружена необходимость перенести существующий столбец, переименовать его и применить к нему общий формат. Вместо того чтобы пытаться выполнить сразу все эти действия, необходимо вначале постараться успешно осуществить операцию “Перемещение столбца” (с. 139), затем реализовать операцию “Переименование столбца” (с. 145) и после этого применить операцию “Введение общего формата” (с. 210), выполняя все эти действия последовательно. Преимущество такого подхода состоит в том, что при возникновении какой-либо ошибки будет легче найти причину нарушения в работе, в связи с тем, что она, по всей вероятности, относится к той части схемы, которая была только что подвергнута изменению.

3.2. Выбор наиболее подходящей операции рефакторинга базы данных

Как описано в настоящей книге, количество операций рефакторинга, которые в принципе могут быть применены к схеме базы данных, весьма велико. Для определения того, какая операция рефакторинга является наиболее подходящей в конкретной ситуации, вначале необходимо проанализировать и понять, с какой задачей вам приходится сталкиваться. В рассматриваемом примере может оказаться так, что Эдуард, впервые обратившись к Анжелике, еще не провел такого анализа. В частности, возможно, Эдуард вначале пришел к выводу, что необходимо обеспечить хранение текущего остатка на счете в таблице `Account`, поэтому требуется ввести новый столбец (с помощью преобразования “Введение нового столбца”, описанного на с. 342), и в связи с этим обратился к

Анжелике только с таким предложением. Но при этом Эдуард не учел, что такой столбец уже существует в таблице `Customer`, которая, безусловно, является для указанного столбца совсем неподходящей; таким образом, Эдуард правильно определил проблему, но предложил неприемлемое решение. На основе своих знаний о существующей схеме базы данных и правильного понимания проблемы, выявленной Эдуардом, Анжелика вместо этого предлагает применить операцию рефакторинга "Перемещение столбца" (с. 139).

Учет наличия других данных

Может оказаться так, что рассматриваемая база данных не является единственным источником данных в организации. Квалифицированный администратор базы данных должен по меньшей мере иметь представление обо всех источниках данных, применяемых на предприятии, а в лучшем случае обладать всеми сведениями по этой теме, поскольку лишь в таком случае он может предложить для работы наилучший источник данных. В рассматриваемом примере может оказаться так, что официально принятым хранилищем информации о счетах, `Account`, в принципе является другая база данных. Если дела обстоят именно так, то перемещение столбца может оказаться неоправданным, поскольку в действительности в этом случае должна быть выполнена операция рефакторинга "Использование официально заданного источника данных" (с. 292).

3.3. Обозначение как устаревшей исходной схемы базы данных

Если доступ к базе данных имеют многочисленные приложения, то может потребоваться организовать работу с учетом того предположения, что не удастся одновременно выполнить данную операцию рефакторинга, а затем снова развернуть все эти программы. Вместо этого необходимо будет предусмотреть переходный период, называемый также периодом поддержки устаревших приложений, для той части исходной схемы, которая подвергается изменениям [**4**; **32**]. В течение переходного периода параллельно поддерживаются и исходная, и новая схема, что позволяет другим группам разработчиков приложений найти время для проведения рефакторинга и повторного развертывания своих систем. Как правило, переходный период продолжается в течение нескольких кварталов или даже лет. В связи с тем, что для полной реализации операции рефакторинга может потребоваться достаточно продолжительное время, возрастает необходимость автоматизации этого процесса в максимально возможной степени. За несколько лет кадровый состав отдела может измениться, а это влечет за собой риск нарушения работы, если часть процесса рефакторинга осуществляется вручную. К сказанному следует добавить, что, даже в том случае, если база данных работает с одним приложением, представителям группы разработчиков может потребоваться переходный период в несколько дней для осуществления всех необходимых действий в специализированной среде интеграции проектов, поскольку разработчики обязаны провести рефакторинг кода и снова протестировать код для оценки готовности к работе с обновленной схемой базы данных.

На рис. 3.3 показан жизненный цикл осуществления операции рефакторинга базы данных в сценарии с несколькими приложениями. Вначале такая операция реализуется в рамках разрабатываемого проекта и в случае успеха в конечном итоге развертывается на

производстве. В течение переходного периода существуют и исходная схема, и новая схема, а для обеспечения полноценной поддержки всех обновлений применяется достаточный объем вспомогательного кода. На протяжении переходного периода необходимо всегда учитывать наличие двух факторов: во-первых, в некоторых приложениях будет использоваться исходная схема, тогда как в других — новая схема, во-вторых, каждое из этих приложений должно работать только с одной, но не с обеими версиями схемы. В рассматриваемом примере некоторые приложения будут работать со столбцом `Customer.Balance`, а другие — со столбцом `Account.Balance`, но не с тем и другим одновременно. Но все эти приложения должны функционировать правильно, независимо от того, с каким столбцом они работают. По истечении переходного периода исходные конструкции схемы и весь вспомогательный код удаляются и выполняется повторное тестирование базы данных. С этого момента следует исходить из предпосылки, что все приложения работают со столбцом `Account.Balance`.

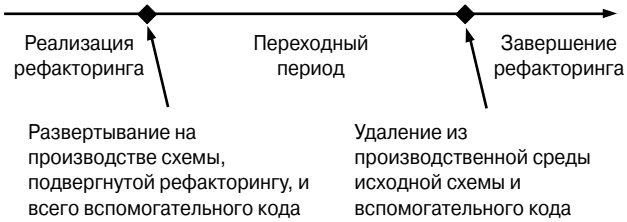


Рис. 3.3. Жизненный цикл операции рефакторинга базы данных в сценарии с несколькими приложениями

На рис. 3.4 показана исходная схема базы данных, а на рис. 3.5 приведен пример того, как может выглядеть схема базы данных в течение переходного периода, который устанавливается на время проведения операции рефакторинга базы данных "Перемещение столбца" применительно к столбцу `Customer.Balance`. На рис. 3.5 изменения выделены полужирным шрифтом (такой стиль используется во всей книге). Обратите внимание на то, что в течение переходного периода поддерживаются обе версии схемы. Столбец `Account.Balance` был добавлен, а столбец `Customer.Balance` отмечен как предназначенный для удаления 14 июня 2006 года или после этой даты. Кроме того, был введен триггер для обеспечения синхронизации значений в обоих столбцах, исходя из того предположения, что новый прикладной код будет работать со столбцом `Account.Balance`, но не будет поддерживать столбец `Customer.Balance` в актуальном состоянии. Аналогичным образом, предполагается, что старый прикладной код, который не был подвергнут рефакторингу в целях использования новых конструкций схемы, не должен обеспечивать поддержку столбца `Account.Balance` в актуальном состоянии. Этот триггер представляет собой пример вспомогательного кода базы данных, т.е. простого и общего кода, без которого невозможно обеспечить целостность базы данных. Этот код должен быть удален в тот же день, что и столбец `Customer.Balance`.

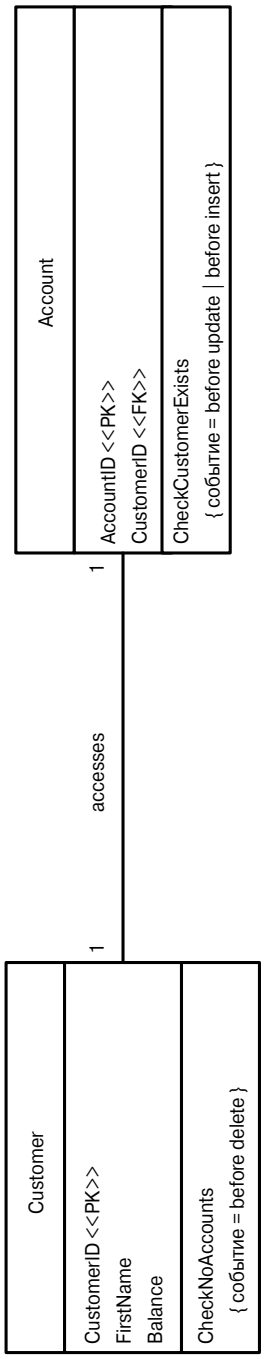


Рис. 3.4. Исходная схема, включающая таблицы `Customer` *и* `Account`

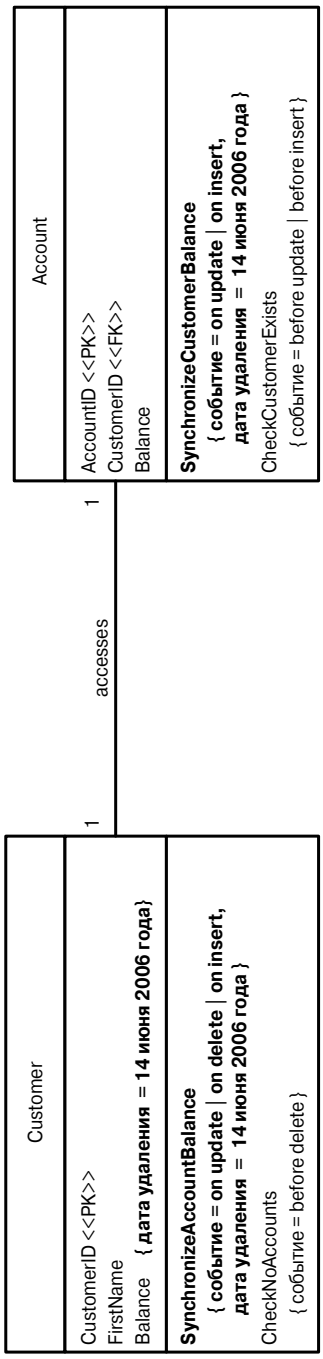


Рис. 3.5. Поддержка обеих версий схемы

Не все операции рефакторинга базы данных требуют переходного периода. Например, переходный период не требуется при осуществлении операций рефакторинга базы данных "Введение ограничения столбца" (с. 207) и "Применение стандартных кодовых обозначений" (с. 188), поскольку эти операции просто улучшают качество данных, сужая набор допустимых значений в столбце.

Стратегии выбора подходящего периода эксплуатации устаревшего программного обеспечения рассматриваются в главе 5 "Стратегии осуществления операций рефакторинга базы данных".

3.4. Тестирование до выполнения, во время выполнения и после выполнения операции рефакторинга

Уверенность в том, что изменения в схему базы данных внесены правильно, появляется, если можно легко убедиться в том, что база данных по-прежнему работает с конкретным приложением после этих изменений, и единственный способ достижения этого состоит в использовании подхода к разработке, основанного на проведении проверок (Test-Driven Development — TDD), который был предложен в главе 1. При использовании подхода на основе TDD разрабатываются тесты, а затем подготавливается лишь такой объем кода, чаще всего на языке определения данных (Data Definition Language — DDL), который позволяет выполнить тест. Разработка в такой форме продолжается до тех пор, пока операция рефакторинга базы данных не будет полностью реализована. Потенциально может потребоваться написать тесты, которые позволяют выполнить перечисленные ниже действия.

- Проверить схему базы данных.
- Проверить способ использования схемы базы данных в приложении.
- Убедиться в правильности переноса данных.
- Проверить код внешних программ.

3.4.1. Проверка схемы базы данных

Операции рефакторинга базы данных могут затронуть схему базы данных, поэтому необходимо подготовить тесты, предназначенные специально для базы данных. Безусловно, на первый взгляд это утверждение может показаться странным, однако действительно существует возможность проверить многие аспекты схемы базы данных, как описано ниже.

- **Хранимые процедуры и триггеры.** Должны быть проверены на таких же основаниях, как и прикладной код.
- **Ограничения ссылочной целостности (Referential integrity — RI).** Кроме того, требуют проверки правила ссылочной целостности, особенно правила каскадного удаления, согласно которым удаляются тесно связанные дочерние строки в случае удаления родительской строки. Правила проверки существования, такие как правила проверки наличия строки с данными о клиенте, соответствующими данным строки банковского счета, должны быть предусмотрены еще до того, как появится

возможность вставки данных в таблицу счетов `Account`, и могут быть также легко протестированы.

- **Определения представлений.** Представления часто используются для реализации важной бизнес-логики. При проверке определений представлений необходимо, в частности, рассмотреть следующие вопросы. Правильно ли действуют алгоритмы поиска и выборки по критериям? Происходит ли выборка правильного количества строк? Происходит ли возврат данных тех столбцов, которые требуются? Находятся ли столбцы и строки в правильном порядке?
- **Заданные по умолчанию значения.** В столбцах часто применяются определенные для них значения, заданные по умолчанию. Должна быть проведена проверка того, действительно ли присваиваются такие значения, предусмотренные по умолчанию. (Нельзя исключить возможность того, что эта часть определения таблицы была случайно удалена.)
- **Инварианты данных.** В столбцах часто применяются инварианты, реализованные в форме ограничений, которые определены для этих столбцов. Например, на столбец с номером может распространяться ограничение, согласно которому он должен содержать только значения от 1 до 7. Эти инварианты требуют проверки.

Для многих специалистов подход, предусматривающий тестирование базы данных, является непривычным, поэтому им приходится сталкиваться с определенными проблемами после принятия методов рефакторинга базы данных в качестве методики разработки, как описано ниже.

- **Недостаточные навыки тестирования.** Эта проблема может быть преодолена благодаря обучению, но может помочь совместная работа со специалистами, обладающими хорошими навыками тестирования (при этом успех достигается даже при объединении усилий АБД без навыков тестирования и специалиста по тестированию без навыков АБД); в крайнем случае можно даже воспользоваться методом проб и ошибок. Важнее всего понять, что специалистам по базам данных эти навыки действительно требуются.
- **Недостаточный ассортимент компонентных тестов для базы данных.** Подход, основанный на тестировании баз данных, принят лишь в немногих организациях, поэтому велика вероятность того, что тестовый набор для существующей схемы окажется недостаточным. Безусловно, такая ситуация не очень благоприятна, но тем весомее становятся стимулы к созданию собственного тестового набора.
- **Недостаточный ассортимент инструментальных средств тестирования базы данных.** К счастью, появляется все больше программного обеспечения с открытым исходным кодом (Open Source Software — OSS), такого как DBUnit (`dbunit.sourceforge.net`), позволяющего управлять тестовыми данными, и SQLUnit (`sqlunit.sourceforge.net`), предназначенного для тестирования хранимых процедур. Кроме того, имеется несколько коммерческих инструментальных средств, предназначенных для тестирования базы данных. Но ко времени написания данной книги оставались еще значительные возможности для поставщиков инструментальных средств по созданию более совершенных инструментов тестирования баз данных.

Итак, рассмотрим, как протестировать изменения в схеме базы данных. Как показано на рис. 3.5, в переходный период в схему базы данных вносятся два изменения, которые необходимо проверить. Первым из них является добавление столбца `Balance` в таблицу `Account`. Это изменение рассматривается в рамках действий по тестированию результатов переноса данных и работы внешней программы, как описано в следующих разделах. Вторым изменением является добавление двух триггеров, `SynchronizeAccountBalance` и `SynchronizeCustomerBalance`, которые, как показывают их имена, обеспечивают синхронизацию двух столбцов с данными. Должны быть предусмотрены такие тесты, которые позволяют гарантировать, что при обновлении столбца `Customer.Balance` обновляется также столбец `Account.Balance`, и наоборот.

3.4.2. Проверка результатов переноса данных

Во многих операциях рефакторинга базы данных требуется выполнить перенос данных, а иногда даже удалить исходные данные. В рассматриваемом примере в ходе реализации операции рефакторинга необходимо скопировать значения данных из столбца `Customer.Balance` в столбец `Account.Balance`. В данном случае должна быть предусмотрена проверка того, действительно ли происходит правильное копирование остатков на счетах отдельных клиентов.

А в таких операциях рефакторинга, как "Применение стандартных кодовых обозначений" (с. 188) и "Осуществление стратегии консолидированных ключей" (с. 197), фактически происходит также удаление части данных. Поэтому требуют проверки и алгоритмы удаления данных. При выполнении первой операции рефакторинга могут быть преобразованы все кодовые значения, такие как `USA` и `U.S.`, в стандартное значение, в данном случае `US`, во всей базе данных. Может потребоваться написать тесты для проверки того, что старые коды больше не используются и что эти коды были преобразованы должным образом в официально утвержденные значения. А при выполнении второй операции рефакторинга может оказаться, что идентификация клиентов в одних таблицах осуществляется по присвоенному им идентификатору клиента, в других — по номеру социального страхования (Social Security Number — SSN), а в третьих — по номеру телефона. Должен быть выбран единый способ идентификации клиентов, возможно, основанный на использовании идентификаторов клиентов, с последующим проведением рефакторинга всех прочих таблиц с тем, чтобы в них вместо прежнего идентификатора применялся столбец этого типа. В данном случае может потребоваться написать тесты для проверки того, что связь между различными строками по-прежнему поддерживается должным образом. (Например, если номер телефона `555-1234` указывает на строку с данными о клиенте `Sally Jones`, то после замены этого номера идентификатором клиента `987654321` ссылка на строку `Sally Jones` должна по-прежнему формироваться правильно.)

3.4.3. Тестирование внешних программ доступа

Как правило, для доступа к базе данных применяется одна или несколько программ, включая те приложения, которые находятся в сфере деятельности специалиста, проводящего изменения в базе данных. Эти программы требуют проверки, как и любые другие информационные активы в организации. Чтобы успешно провести рассматриваемую операцию рефакторинга базы данных, необходимо обладать правами на внедрение окончательной схемы, показанной на рис. 3.6, и иметь возможность определить, возникают ли

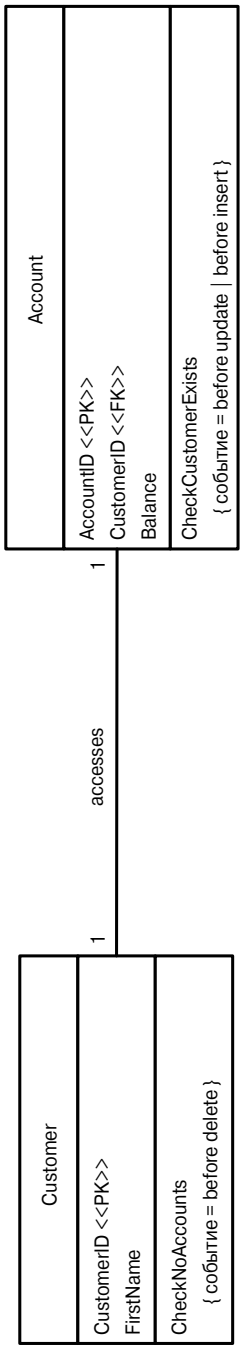


Рис. 3.6. Окончательная версия схемы базы данных

какие-либо нарушения в работе внешних программ доступа. Уверенность в том, что операция рефакторинга схемы базы данных проведена успешно, достигается лишь при одном условии — при наличии полного регрессионного тестового набора для тех программ, в которых используется база данных, однако подобные тестовые наборы предусмотрены далеко не в каждой организации. Еще раз отметим, что указанная ситуация создает самые лучшие предпосылки для проведения работ по созданию собственного тестового набора. Авторы рекомендуют приступить к созданию сразу всего тестового кода, требуемого для поддержки каждой отдельной операции рефакторинга базы данных для всех внешних программ доступа. (Но в действительности эти тесты должны написать владельцы прикладных систем, а не специалисты по базам данных.) Если читатель прислушается к этим рекомендациям, то со временем получит весь требуемый тестовый набор.

3.5. Модификация схемы базы данных

Рассмотрим ситуацию, в которой совместно работают два разработчика, Эдуард и Анжелика, чтобы внести изменения в своей специализированной среде разработки. Как показано на рис. 3.5, задача состоит в том, чтобы ввести дополнительный столбец `Account.Balance`, а также добавить два триггера, `SynchronizeAccountBalance` и `SynchronizeCustomerBalance`. Код DDL, предназначенный для выполнения указанных действий, показан ниже.

```
ALTER TABLE Account ADD Balance Numeric;
COMMENT ON Account.Balance 'Move of Customer.Balance column, finaldate
= 2006-06-14';

CREATE OR REPLACE TRIGGER SynchronizeCustomerBalance
 BEFORE INSERT OR UPDATE
 ON Account
 REFERENCING OLD AS OLD NEW AS NEW
 FOR EACH ROW
 DECLARE
 BEGIN
  IF :NEW.Balance IS NOT NULL THEN
   UpdateCustomerBalance;
  END IF;
 END;
/
COMMENT ON SynchronizeCustomerBalance 'Move of Customer.Balance column
to Account, dropdate = 2006-06-14';

CREATE OR REPLACE TRIGGER SynchronizeAccountBalance
 BEFORE INSERT OR UPDATE OR DELETE
 ON Customer
 REFERENCING OLD AS OLD NEW AS NEW
 FOR EACH ROW
 DECLARE
 BEGIN
  IF DELETING THEN
   DeleteCustomerIfAccountNotFound;
  END IF;
```

```
  IF (UPDATING OR INSERTING) THEN
   IF :NEW.Balance IS NOT NULL THEN
    UpdateAccountBalanceForCustomer;
   END IF;
  END IF;
 END;
/
COMMENT ON SynchronizeAccountBalance 'Move of Customer.Balance column
to Account, dropdate = 2006-06-14'
```

Ко времени написания данной книги не было предусмотрено ни одного автоматизированного инструментального средства выполнения операций рефакторинга базы данных, поэтому весь приведенный в этом примере код пришлось разрабатывать вручную. Но не следует беспокоиться. Такая ситуация со временем изменится. Однако на данный момент необходимо написать единственный сценарий, содержащий приведенный выше код, который можно применить к рассматриваемой схеме базы данных. Авторы рекомендуют присваивать каждому сценарию уникальный, инкрементный номер. Этого можно добиться проще всего, начав со сценария номер один и увеличивая значение счетчика после определения каждой новой операции рефакторинга базы данных; самый легкий способ осуществления этого состоит в использовании номера сборки рассматриваемого приложения. Но для успешного осуществления такой стратегии в среде с несколькими коллективами разработчиков необходимо предусмотреть какой-то способ назначения номеров, которые остаются уникальными во всех этих коллективах, или же присваивать уникальный идентификатор группы разработчиков отдельным операциям рефакторинга. По сути, необходимо иметь возможность отличить операцию рефакторинга номер 1701 группы `A` от операции рефакторинга номер 1701 группы `B`. Другой вариант, который рассматривается более подробно в главе 5, предусматривает присваивание операциям рефакторинга временны*х* отметок.

При этом следует применять небольшие сценарии, относящиеся к отдельным операциям рефакторинга, по причинам, описанным ниже.

- **Простота.** Небольшие, целенаправленные сценарии проще сопровождать, чем сценарии, состоящие из многих этапов. К тому же, если будет обнаружено, что какую-то операцию рефакторинга не следует выполнять в связи с возникновением непредвиденных проблем (возможно, пока не удается обновить важное приложение, которое получает доступ к изменяющейся части схемы), то, например, можно будет легко отказаться от осуществления этой операции рефакторинга.
- **Правильность.** Необходимо предусмотреть возможность применения каждой операции рефакторинга в соответствующем порядке к схеме базы данных, чтобы обеспечить развитие этой схемы определенным образом. Операции рефакторинга могут следовать одна за другой. Например, может быть переименован какой-то столбец, а затем, через несколько недель, перемещен в другую таблицу. В таком случае вторая операция рефакторинга будет зависеть от первой операции рефакторинга, поскольку в коде, применяемом для ее выполнения, должно быть указано новое имя столбца.
- **Учет наличия разных версий.** Может оказаться так, что в различных экземплярах базы данных применяются разные версии схемы базы данных.

Например, в специализированной среде разработки Эдуарда может применяться версия 163, в специализированной среды интеграции проектов — версия 161, в специализированной среде контроля качества и тестирования — версия 155, а в производственной базе данных — версия 134. Для перевода схемы, применяемой в специализированной среде интеграции проектов, на версию 163 необходимо просто провести операции рефакторинга базы данных с номерами 162 и 163. Для отслеживания номеров версий требуется создать общую таблицу, допустим, `DatabaseConfiguration`, в которой, кроме всего прочего, хранится текущий номер версии.

Эта таблица более подробно описана в главе 5.

Ниже приведен код DDL, который должен быть выполнен по отношению к базе данных после завершения переходного периода (как описано в главе 4). В соответствии с приведенной выше рекомендацией, этот код должен быть представлен в виде единственного файла сценария, в данном случае с указанием идентификатора 163, и применен должным образом к схеме базы данных с учетом сложившейся последовательности.

```
ALTER TABLE Customer DROP COLUMN Balance;
DROP TRIGGER SynchronizeAccountBalance;
DROP TRIGGER SynchronizeCustomerBalance;
```

Соблюдение соглашений по проектированию базы данных

Важная составляющая реализации любой операции рефакторинга состоит в обеспечении того, чтобы даже изменившаяся часть схемы базы данных соответствовала принятым в конкретной организации рекомендациям по разработке базы данных. Такие рекомендации должны подготавливаться и сопровождаться группой администрирования базы данных и, как минимум, должны содержать инструкции по адресованию, именованию и документированию.

3.6. Перенос исходных данных

При проведении многих операций рефакторинга базы данных возникает необходимость проводить каким-то образом манипуляции с исходными данными. Например, иногда может лишь потребоваться переместить данные из одного местоположения в другое, для чего применяется операция “Перемещение данных” (с. 219). В других случаях возникает необходимость провести очистку самих значений данных; такие ситуации обычно связаны с проведением операций рефакторинга качества данных (они описаны в главе 7 “Операции рефакторинга качества данных”), таких как “Применение стандартного типа” (с. 192) и “Введение общего формата” (с. 210).

По аналогии с тем, как проводятся модификации схемы базы данных, потенциально может потребоваться создать сценарий для выполнения требуемого переноса данных. Этому сценарию также должен присваиваться идентификационный номер, как и любым другим сценариям, поскольку это позволяет упростить его сопровождение. В рассматриваемом примере перемещения столбца `Customer.Balance` в таблицу `Account` сценарий переноса данных должен содержать следующий код на языке манипулирования данными (Data Manipulation Language — DML):

```
/*
Одноразовый перенос данных из таблицы Customer.Balance в таблицу
Account.Balance
*/

UPDATE Account SET Balance =
 (SELECT Balance FROM Customer
  WHERE CustomerID = Account.CustomerID);
```

В зависимости от качества существующих данных может вскоре обнаружиться потребность в дополнительной очистке исходных данных. Для этого необходимо применить одну или несколько операций рефакторинга качества данных базы данных. Тем не менее, занимаясь проведением структурных и архитектурных операций рефакторинга базы данных, следует на время отложить в сторону решение проблем повышения качества данных. Проблемы качества данных часто возникают в связи с использованием устаревших проектов баз данных, в которых со временем из-за отсутствия должного контроля ухудшилось качество данных.

Необходимость в документировании, которая отражает потребность в проведении рефакторинга

Если обнаруживается необходимость написать сопроводительную документацию, содержащую сведения о таблице, столбце или хранимой процедуре, это чаще всего является наглядным свидетельством чрезмерной усложненности соответствующей части схемы базы данных, иными словами, потребности в проведении рефакторинга этой части схемы. Возможно, простое переименование объекта позволит обойтись без написания нескольких абзацев документации. Чем понятнее проект, тем меньше документации требуется для его описания.

3.7. Рефакторинг внешних программ доступа

В связи с внесением изменений в схему базы данных часто возникает необходимость проведения рефакторинга всех тех существующих внешних программ, которые получают доступ к изменившейся части схемы. Как описано в главе 2 “Операции рефакторинга базы данных”, к этому относятся существующие приложения, инфраструктуры доступа к базе данных, код репликации данных, системы формирования отчетов и многие другие программы.

Рекомендации по эффективному проведению операций рефакторинга внешних программ доступа широко представлены в литературе; в качестве примера можно указать следующие книги.

- *Refactoring: Improving the Design of Existing Code* [**17**] является классической книгой по этой теме.
- В книге *Working Effectively with Legacy Code* [**16**] описано, как подвергнуть рефакторингу устаревшие системы, которые существовали в организации в течение многих лет.

- Книга *Рефакторинг с использованием шаблонов* [**23**] посвящена описанию способов методического рефакторинга кода, позволяющих реализовать объединенный проект и применить архитектурные шаблоны.

Если количество программ, получающих доступ к базе данных, достаточно велико, возникает риск того, что некоторые из этих программ не будут обновлены ответственными за них группами разработчиков, или, что еще хуже, за ними в настоящий момент могут быть даже не закреплены какие-либо разработчики. Из этого следует, что на кого-то должна быть возложена ответственность за обновление соответствующего приложения (приложений), а также ответственность за выделение необходимых ресурсов. В благоприятных обстоятельствах за работу внешних программ отвечают отдельные группы разработчиков; в противном случае обязанности по внесению требуемых изменений должна взять на себя группа специалистов, осуществляющая операцию рефакторинга базы данных. Бывает очень досадно, когда обнаруживаешь, что организационные проблемы, которые приходится преодолевать, добиваясь обновления других систем, часто намного перевешивают технические сложности, связанные с осуществлением такого обновления.

Цель непрерывной разработки

В идеальном случае в любой организации должна непрерывно осуществляться работа над всеми применяемыми в ней приложениями, что позволяет развивать их со временем и регулярно развертывать усовершенствованные версии. Безусловно, на первый взгляд такая цель кажется труднодостижимой, и действительно может оказаться такой, но разве отдел информационной технологии не должен стремиться к тому, чтобы все системы в организации постоянно соответствовали ее изменяющимся потребностям? В подобных вариантах среды может быть предусмотрен относительно короткий переходный период, поскольку известно, что все приложения, получающие доступ к базе данных, постоянно развиваются и поэтому могут быть быстро обновлены для работы с изменившейся схемой.

Итак, что же можно сделать, если не предусмотрено финансирование каких-либо обновлений внешних программ? Для решения этой задачи можно выбрать одну из двух основных стратегий. Во-первых, можно провести операцию рефакторинга базы данных и назначить для нее переходный период в несколько десятилетий. Это позволяет обеспечить дальнейшую эксплуатацию внешних программ, которые невозможно изменить, но в других приложениях может быть предусмотрен доступ к улучшенному проекту. К сожалению, эта стратегия имеет один недостаток, связанный с тем, что вспомогательный код, предназначенный для поддержки обеих схем, должен применяться в течение долгого времени, снижая производительность базы данных и загромождая базу данных. Во-вторых, можно просто отказаться от проведения операции рефакторинга.

3.8. Выполнение регрессионных тестов

Одной из составляющих работы по реализации операций рефакторинга является тестирование в целях проверки правильности их выполнения. Как было указано выше, необходимо вносить небольшие изменения, тестировать эти изменения, затем вносить следующие изменения, снова их тестировать и т.д., пока операция рефакторинга не будет

выполнена. При этом следует стремиться обеспечить автоматизацию всех операций тестирования в максимально возможной степени. Существенное преимущество подхода, основанного на использовании операций рефакторинга базы данных, состоит в том, что каждая из этих операций представляет собой небольшое изменение, поэтому в случае неудачного завершения теста складывается довольно полное представление о том, в чем заключается проблема, — в том, где было только что внесено изменение.

3.9. Применение в работе средств контроля версий

После того как операция рефакторинга базы данных оканчивается успешно, необходимо внести все результаты выполненной работы в систему контроля над внесением изменений в конфигурацию (Configuration Management — CM), применяя для этого инструментальное средство управления версиями. Если все программное обеспечение, предназначенное для сопровождения базы данных, обслуживается по такому же принципу, как и весь прочий исходный код, то указанная рекомендация является вполне осуществимой. К программному обеспечению, которое должно быть охвачено средствами управления версиями, относится следующее.

- Любые созданные сценарии.
- Тестовые данные и (или) код формирования тестовых данных.
- Тестовые примеры.
- Документация.
- Модели.

3.10. Объявление о проведении операции рефакторинга

База данных — это общий ресурс. Как минимум, ее приходится использовать совместно с группой разработчиков приложений или даже с несколькими группами. Поэтому необходимо сообщить всем заинтересованным сторонам, что была проведена операция рефакторинга базы данных. На ранних этапах жизненного цикла операции рефакторинга необходимо сообщить об изменениях представителям группы специалистов, в рамках которой проводится эта операция; для этого может оказаться достаточным объявить об изменении на следующем плановом собрании представителей группы. В среде базы данных с несколькими приложениями необходимо сообщить об изменениях представителям других групп; это особенно важно, если принято решение перенести операцию рефакторинга в среду тестирования и подготовки производства. Для этого может оказаться достаточным отправить по электронной почте письмо во внутренний список рассылки, специально используемый для объявлений об изменениях в базе данных; еще один вариант состоит в том, что можно включить отдельный пункт в регулярный отчет о состоянии разработки проекта или составить формальный отчет для передачи в группу администраторов, эксплуатирующих базу данных.

Важной составляющей действий, связанных с публикацией сведений о предстоящих изменениях, является обновление всей соответствующей документации. Наличие такой документации становится очень важным, когда приходится предпринимать усилия по

продвижению и развертыванию предстоящих изменений (подробнее об этом — в главе 4), поскольку все другие группы специалистов должны знать, как развивается схема базы данных. Для этого можно проще всего подготовить примечания к выпуску базы данных, в которых подытожены сведения об осуществляемых изменениях и последовательно перечислены все операции рефакторинга базы данных. В рассматриваемом примере операция рефакторинга базы данных могла бы быть представлена в этом списке как "163: перемещение столбца `Customer.Balance` в таблицу `Account`". Такие примечания к выпуску, скорее всего, потребуются администраторам предприятия, чтобы они могли обновить соответствующие метаданные. (А лучше всего провести операцию обновления метаданных силами специалистов, осуществляющих операцию рефакторинга, в составе этой операции.)

Может также потребоваться обновить физическую модель данных (Physical Data Model — PDM), относящуюся к базе данных. Такая PDM представляет собой основную модель, описывающую схему базы данных, и чаще всего является одной из нескольких основополагающих моделей, созданных в проектах разработки приложений; поэтому должна поддерживаться на актуальном уровне в максимально возможной степени.

Отказ от преждевременной публикации модели данных

Объектная схема и схема базы данных чаще всего подвергаются изменениям на первых порах, поскольку при любом эволюционном подходе к разработке проект со временем развивается. Учитывая такую первоначальную изменчивость, необходимо дождаться, пока новые части схемы не станут стабильными, и лишь после этого опубликовать информацию о том, какие произошли изменения в физической модели данных. Это позволяет уменьшить трудоемкость сопровождения документации, а также свести к минимуму отрицательное влияние на работу других групп разработчиков приложений, которые пользуются сведениями о схеме базы данных, предоставляемыми специалистами по базам данных.

3.11. Резюме

Наиболее трудоемкая часть работы по подготовке операции рефакторинга базы данных выполняется в специализированной среде разработки; при этом оптимальные результаты достигаются в том случае, если разработчики привлекают к работе администратора базы данных. На первом этапе необходимо проверить, целесообразно ли вообще применять операцию рефакторинга базы данных, ведь может оказаться, что в настоящее время затраты, связанные с проведением операции рефакторинга, перевешивают преимущества, или что текущая схема является наилучшим проектом для данной конкретной задачи. А если операция рефакторинга действительно требуется, то для достижения намеченной цели необходимо выбрать наиболее подходящую операцию. В среде с несколькими приложениями при осуществлении многих операций рефакторинга необходимо обеспечить параллельную эксплуатацию и первоначальной, и новой версий схемы в течение переходного периода, который должен быть достаточно продолжительным, для того чтобы была возможность провести доработку всех приложений, получающих доступ к этой части схемы.

Для реализации операции рефакторинга следует руководствоваться подходом на основе первоочередного тестирования, поскольку он позволяет увеличить шансы обнаружения любых нарушений в работе, вызванных проведением операции рефакторинга. Требуется внести изменения в схему базы данных, в случае необходимости переместить все затрагиваемые изменениями исходные данные, а затем откорректировать все внешние программы, которые получают доступ к схеме. Вся указанная работа должна быть организована на основе управления версиями; после реализации операции рефакторинга в среде разработки необходимо объявить об этом членам собственного коллектива, а затем сообщить об этом всем заинтересованным представителям внешних организаций, которым, возможно, потребуется информация об изменении схемы.

Глава 4

Развертывание на производстве

Если мы будем продолжать двигаться в намеченном направлении, то вскоре достигнем цели.

Д-р Ирвин Кори

Недостаточно просто подвергнуть рефакторингу схемы базы данных в специализированной среде разработки и интеграции. Необходимо также развернуть изменения на производстве. Применяемый для этого способ должен отражать особенности существующего процесса развертывания в организации; может также потребоваться усовершенствовать этот процесс, чтобы он соответствовал эволюционному подходу, описанному в настоящей книге. Может оказаться так, что в конкретной организации уже имеется опыт развертывания изменений в базах данных, однако, поскольку изменения в схеме часто рассматриваются с опасением, может быть также обнаружено, что опыт, накопленный в этой области, явно недостаточен. Настало время изменить данную ситуацию.

Указанному стремлению способствует то, что развертывание операций рефакторинга базы данных осуществляется намного более безопасно по сравнению с развертыванием традиционных изменений в схеме базы данных, разумеется, при условии выполнения рекомендаций, приведенных в настоящей книге. Это утверждение авторов действительно оправдывается по нескольким причинам. Во-первых, развертывание отдельных операций рефакторинга базы данных связано с меньшим риском по сравнению с развертыванием традиционных изменений в схеме базы данных, поскольку такие операции меньше по объему и проще. Но при осуществлении на практике развертывания целого ряда операций рефакторинга базы данных ситуация может стать сложнее, если не обеспечен должный контроль над этими операциями. В настоящей главе приведены рекомендации по решению именно этой задачи. Во-вторых, при использовании подхода на основе разработки, управляемой тестами (Test-Driven Development — TDD), должен быть предусмотрен полный регрессионный тестовый набор, позволяющий проверить правильность осуществления операций рефакторинга. А поскольку становится известно, что операции рефакторинга действительно выполняются успешно, их развертывание может осуществляться с большей уверенностью. В-третьих, в связи с наличием переходного периода, в течение которого и старые, и новые схемы существуют параллельно, не требуется слишком спешить с развертыванием изменений в приложениях, отражающих изменения в схеме.

Для успешного развертывания операций рефакторинга базы данных на производстве необходимо выработать стратегии осуществления указанных ниже действий.

- Эффективное развертывание с передачей из одной специализированной среды в другую.
- Применение наборов операций рефакторинга базы данных.
- Планирование подходящих интервалов развертывания.
- Развертывание всей системы.
- Удаление устаревшей схемы.

4.1. Эффективное развертывание с передачей из одной специализированной среды в другую

В главе 1 "Эволюционная разработка базы данных" описан подход, в котором предусмотрено применение целого ряда специализированных вариантов среды, предназначенных для реализации, тестирования и эксплуатации конкретных систем. Как показано на рис. 4.1, каждая группа проектировщиков имеет в своем распоряжении целый ряд вариантов специализированной среды разработки и, возможно, даже собственную демонстрационную специализированную среду, соответствующую конкретным требованиям. С другой стороны, специализированная среда тестирования, применяемая на этапе подготовки производства, является общей для всех групп разработчиков, как и производственная среда. На этом рисунке также показано, что между различными вариантами специализированной среды предусмотрены шлюзы развертывания. Задача развертывания операций рефакторинга базы данных, связанная с переносом этих операций с рабочей станции разработчика в общую специализированную среду интеграции проектов, является относительно простой, поскольку возникновение какой-либо ошибки не приводит к серьезным последствиям, так как влияние этой ошибки распространяется только на одну группу разработчиков. Но задача развертывания в одной или нескольких вариантах среды тестирования для подготовки к внедрению на производстве, как правило, немного сложнее, поскольку влияние ошибок становится гораздо более весомым. Ошибки могут не только приводить к тому, что система станет недоступной для лиц, проводящих тестирование, но и стать причиной аварийного завершения работы других систем в этих вариантах среды и таким образом оказать отрицательное воздействие на работу других групп разработчиков. Развертывание изменений на производстве часто представляет собой процесс, требующий строгого контроля, поскольку потенциальная стоимость ошибки весьма велика из-за возможного нарушения работы клиентов.

Чтобы обеспечить развертывание изменений в каждой специализированной среде, необходимо не только создать приложение, но и выполнить сценарии управления базой данных (журналом регистрации изменений, журналом регистрации обновлений и журналом регистрации операций переноса данных, или же эквивалентными объектами; см. главу 3). На следующем этапе необходимо повторно запустить регрессионные тесты, чтобы убедиться в том, что система по-прежнему работает; в противном случае необходимо внести исправления в специализированной среде разработки, снова выполнить развертывание и провести повторное тестирование. Назначение работы, выполняемой

в специализированной среде интеграции проектов, состоит в том, чтобы проверить правильность изменений, подготовленных отдельными разработчиками (или группами разработчиков), а в специализированной среде тестирования/обеспечения качества на этапе подготовки производства назначение выполняемой работы состоит в проведении проверки качества функционирования конкретной системы в сочетании с другими системами в организации.

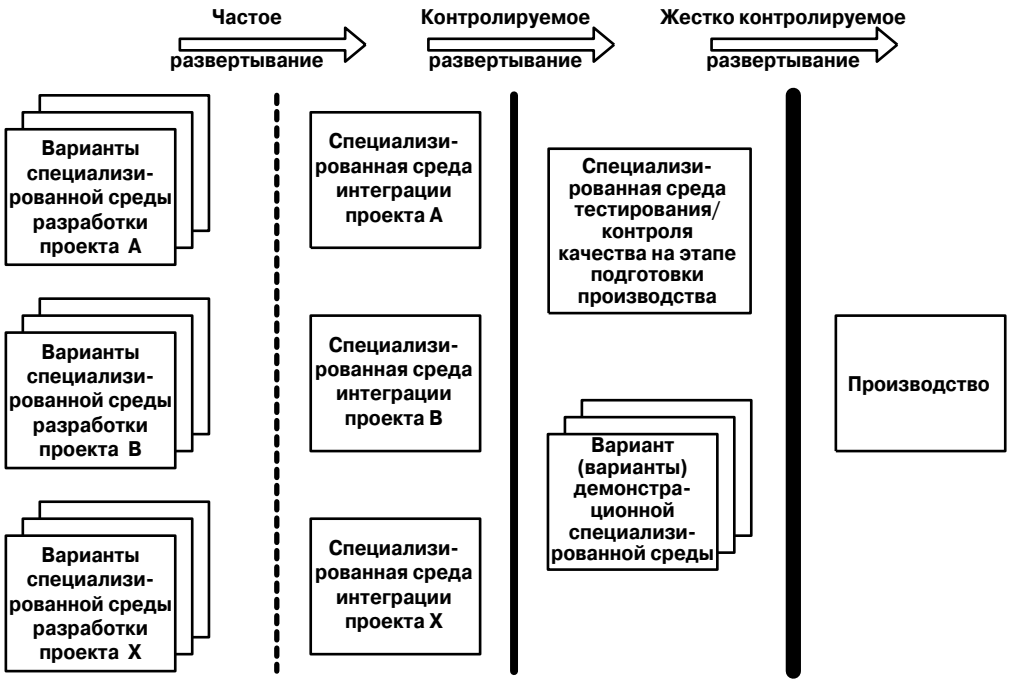


Рис. 4.1. Развертывание с передачей из одной специализированной среды в другую

Нередко приходится видеть, как разработчики переносят изменения, подготовленные в своей специализированной среде разработки, в специализированную среду интеграции проектов несколько раз в день. А что касается группы разработчиков, то ее представители должны стремиться обеспечить развертывание своей системы хотя бы в демонстрационной среде по меньшей мере один раз на протяжении цикла разработки, чтобы иметь возможность узнать мнение представителей самой организации, заинтересованных в разработке проекта, о предлагаемом в настоящее время варианте работоспособного программного обеспечения. Еще лучше иметь возможность развернуть свою систему в среде тестирования на этапе подготовки производства, чтобы проверить ее соответствие предъявленным требованиям и провести тестирование всей системы; в идеальном случае такое мероприятие следует провести по крайней мере один раз на протяжении цикла разработки. Регулярное развертывание в среде подготовки производства должно проводиться по двум указанным ниже причинам. Во-первых, это позволяет получить конкретные результаты обратной связи, касающиеся того, насколько успешно функционирует систе-

ма в действительности. Во-вторых, регулярное развертывание способствует накоплению опыта применения успешных методов развертывания; в связи с тем, что сценарии инсталляции выполняются достаточно часто и при этом неоднократно происходит проверка их правильности, очень быстро достигается такое положение, что развертывание системы на производстве может быть осуществлено вполне успешно.

4.2. Применение наборов операций рефакторинга базы данных

В наши дни работа групп разработчиков организована в виде коротких итераций; например, в группах разработчиков, взявших на вооружение адаптивный подход, весьма часто применяются итерации, которые занимают не больше одной или двух недель. Таким образом, группа разработчиков создает полностью функциональное программное обеспечение почти через каждую неделю, но это не означает, что на производстве происходит развертывание новой версии системы раз в неделю. Как правило, развертывание новых версий на производстве происходит один раз в несколько месяцев. Из этого следует, что группы разработчиков вынуждены комплектовать все операции рефакторинга базы данных, подготовленные ими со времени последнего развертывания изменений на производстве, чтобы иметь возможность применить эти операции должным образом.

Как описано в главе 3, такую задачу можно проще всего решить, рассматривая каждую отдельную операцию рефакторинга базы данных как собственную транзакцию, которая должна быть реализована как совокупность сценариев на языке определения данных (Data Definition Language — DDL), позволяющих должным образом изменить схему базы данных и перенести все необходимые данные, а также внести изменения в исходный код программ, которые получают доступ к соответствующей части схемы. Этой транзакции должен быть присвоен уникальный идентификатор, как правило, числовой или представляющий собой отметку даты/времени, который позволяет контролировать проведение операций рефакторинга. Благодаря этому появляется возможность применить операции рефакторинга последовательно, включив их либо в подготовленный вручную сценарий, либо в какое-то универсальное инструментальное средство, и тем самым провести их развертывание в любой необходимой специализированной среде.

При этом нельзя исходить из предположения, что все схемы базы данных идентичны друг другу, поэтому должен быть предусмотрен какой-то способ безопасного применения различных сочетаний операций рефакторинга к каждой схеме. Например, рассмотрим схему, в которой в качестве идентификаторов используются числа (рис. 4.2). На этом рисунке приведены различные базы данных разработчиков, каждая из которых имеет разную версию. Одна база данных имеет версию 813, другая — 811, а третья — 815. База данных интеграции проектов имеет версию 811, поэтому можно сделать вывод, что одна схема базы данных изменилась со времени ее последней синхронизации со средой интеграции проектов, еще один разработчик не модернизировал свою версию схемы и еще не получил наиболее современную версию (поскольку 809 меньше, чем 811), а третий разработчик вносит изменения одновременно с другими разработчиками. Все изменения должны быть внесены практически одновременно, поскольку в ином случае ни один из разработчиков не сможет передать предложенные им изменения в среду интеграции (ведь иначе среда будет иметь такой же идентификационный номер, как и одна из двух баз данных). Для обновления базы данных интеграции проектов необходимо провести в ней изменения, начиная с номера 812; чтобы обновить базу данных среды тестирования для

подготовки производства, требуется начать со сценария обновления 806; обновление демонстрационной среды должно начинаться со сценария обновления 801; обновление производственной базы данных должно начинаться с изменения, имеющего номер 758. Более того, не следует вносить все изменения во все версии, например, к следующему выпуску производственная версия может быть доведена только до номера 794.

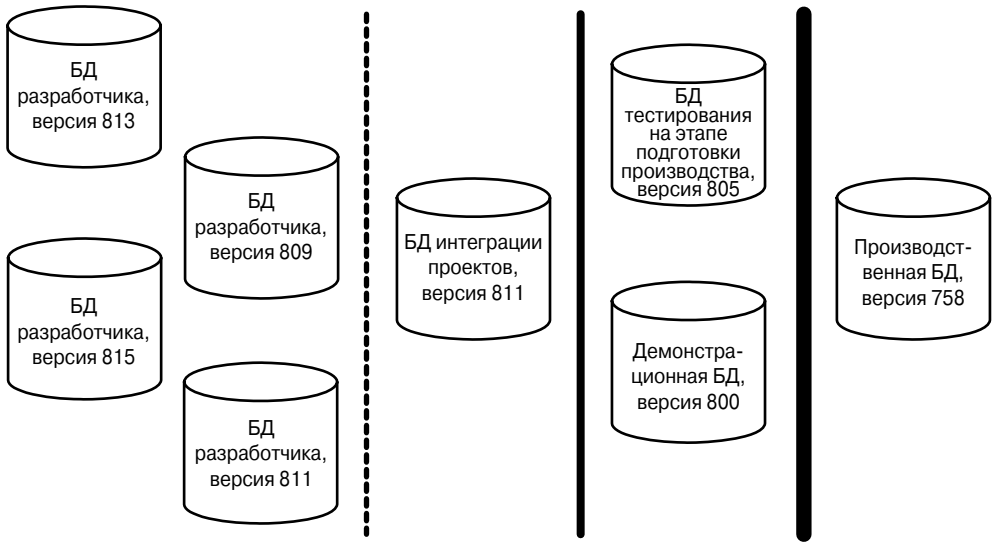


Рис. 4.2. Подготовка разных версий для различных баз данных

Проще всего можно рассматривать эту ситуацию так, что с начала разработки нового выпуска системы создается новый стек операций рефакторинга базы данных. В ходе осуществления усилий по разработке в стек продолжают добавляться новые изменения в схеме, а иногда некоторые из них удаляются, если обнаруживается, что от этих изменений следует отказаться. По завершении разработки текущего выпуска стек изменений закрывается, после чего рассматривается как набор операций изменения схемы для текущего выпуска.

4.3. Планирование подходящих интервалов развертывания

Для развертывания системы на производстве выбирается определенный период времени, называемый *интервалом развертывания*, или, чаще всего *подходящим интервалом выпуска*. Как правило, эксплуатационный персонал руководствуется строгими правилами, касающимися того, какое время может применяться группами разработчиков приложений для развертывания систем. Часто можно наблюдать такую ситуацию, что группы разработчиков приложений получают разрешение на развертывание новых выпусков только в субботу от 14:00 до 18:00, а на исправление ошибок отводится время с 16:00 до 18:00 на следующий день. Такие подходящие интервалы развертывания обычно совпадают с периодами снижения активности системы. Кроме того, могут быть преду-

смотрены правила, касающиеся предоставления разрешений на внесение изменений в схему базы данных, например только в третью субботу каждого месяца. В небольших организациях, особенно если в них не осуществляется одновременно несколько проектов разработки, может быть принято решение о проведении развертывания изменений только один раз в несколько месяцев.

Из этого следует то, что коллективу разработчиков не будет разрешено развертывать новую версию системы на производстве после каждого ее выпуска; вместо этого должно быть запланировано развертывание в заранее определенный подходящий интервал развертывания. На рис. 4.3 условно представлена эта концепция и показано, как две группы разработчиков планируют развертывание предложенных ими изменений (включая операции рефакторинга базы данных) в предусмотренные для этого интервалы развертывания. В одних случаях изменения, подлежащие развертыванию, отсутствуют, в других изменения могут быть предложены только одной группой разработчиков, а в третьих изменения могут быть подготовлены к развертыванию обеими группами разработчиков. Подходящие интервалы развертывания (см. рис. 4.3) совпадают с периодами окончательного развертывания и передачи из среды тестирования на этапе подготовки производства в производственную среду (см. рис. 4.1).

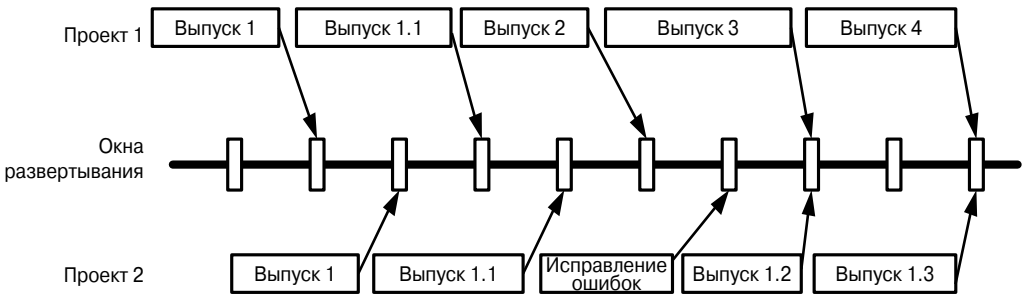


Рис. 4.3. Развертывание изменений схемы на производстве в заранее определенные интервалы времени

Безусловно, группы разработчиков должны координировать свою деятельность со всеми другими группами, которые осуществляют развертывание в течение того же подходящего интервала развертывания. Такая координация должна быть проведена задолго до перехода к развертыванию, и следует учитывать, что фактически основная причина, по которой применяется среда тестирования на этапе подготовки к производству, состоит в том, чтобы обеспечить работу в специализированной среде, в которой могут быть устранены проблемы взаимодействия нескольких разных систем. Независимо от того, сколько операций рефакторинга базы данных должно быть применено к производственной базе данных или сколько групп разработчиков участвовало в подготовке операций рефакторинга, все эти операции должны быть вначале проверены в среде тестирования на этапе подготовки к производству и лишь затем применены на производстве.

Основное преимущество использования заранее принятых подходящих интервалов развертывания состоит в том, что они обеспечивают возможность создания механизма управления тем, что и когда поступает на производство. Благодаря этому эксплуатационный персонал получает возможность организовать свое время, группы разработчиков

планируют свою деятельность с учетом предоставления готовых результатов в определенные даты, а конечные пользователи могут рассчитывать на то, что необходимые им новые функциональные средства будут предоставлены в назначенное время.

4.4. Развертывание всей системы

Развертывание операций рефакторинга базы данных, отдельно взятых, как правило, не производится. Вместо этого развертывание этих операций осуществляется в составе общих действий по развертыванию одной или нескольких систем. Развертывание осуществляется проще, если обновлению подлежат одно приложение и одна база данных, и такая ситуация действительно встречается на практике. Но более реалистичным является предположение, согласно которому обычно приходится рассматривать ситуацию, в которой развертывается одновременно несколько систем и несколько источников данных. На рис. 4.4 представлена блок-схема UML, на которой условно изображен процесс развертывания. Как показано на этом рисунке, необходимо осуществить описанные ниже действия.

1. **Резервное копирование базы данных.** Безусловно, решение этой задачи применительно к крупным производственным базам данных часто сопряжено со значительными сложностями, но требования по созданию полной резервной копии являются почти обязательными, поскольку они позволяют восстановить базу данных до приемлемого состояния, в том случае, если попытка развертывания окончится неудачей. Одним из преимуществ операций рефакторинга базы данных является то, что они невелики, поэтому каждая из этих операций может быть отменена отдельно от других, но если развертывание операций рефакторинга осуществляется достаточно редко, это преимущество теряется, поскольку повышается вероятность того, что одни операции рефакторинга будут зависеть от других.
2. **Выполнение заранее подготовленных регрессионных тестов.** Вначале необходимо убедиться в том, что существующая производственная система (системы) действительно работоспособна и функционирует должным образом. Безусловно, этого не должно происходить, но кто-то может неумышленно внести изменение, о котором не будет известно другим разработчикам. Если выполнение тестового набора, соответствующего предыдущему варианту развертывания, окончится неудачей, то лучше всего отказаться от применения текущего варианта развертывания и выяснить, в чем состоит проблема. Следует отметить, что при подготовке регрессионных тестов необходимо следить за тем, чтобы они не создавали каких-либо непреднамеренных побочных эффектов в производственной среде. Из этого следует, что и деятельность по тестированию требует соблюдения мер предосторожности.
3. **Развертывание приложения (приложений), в которое были внесены изменения.** При этом необходимо руководствоваться регламентированными процедурами.
4. **Развертывание операций рефакторинга базы данных.** Для этого необходимо выполнить соответствующие сценарии модификации схемы и сценарии переноса данных применительно к конкретным источникам данных.

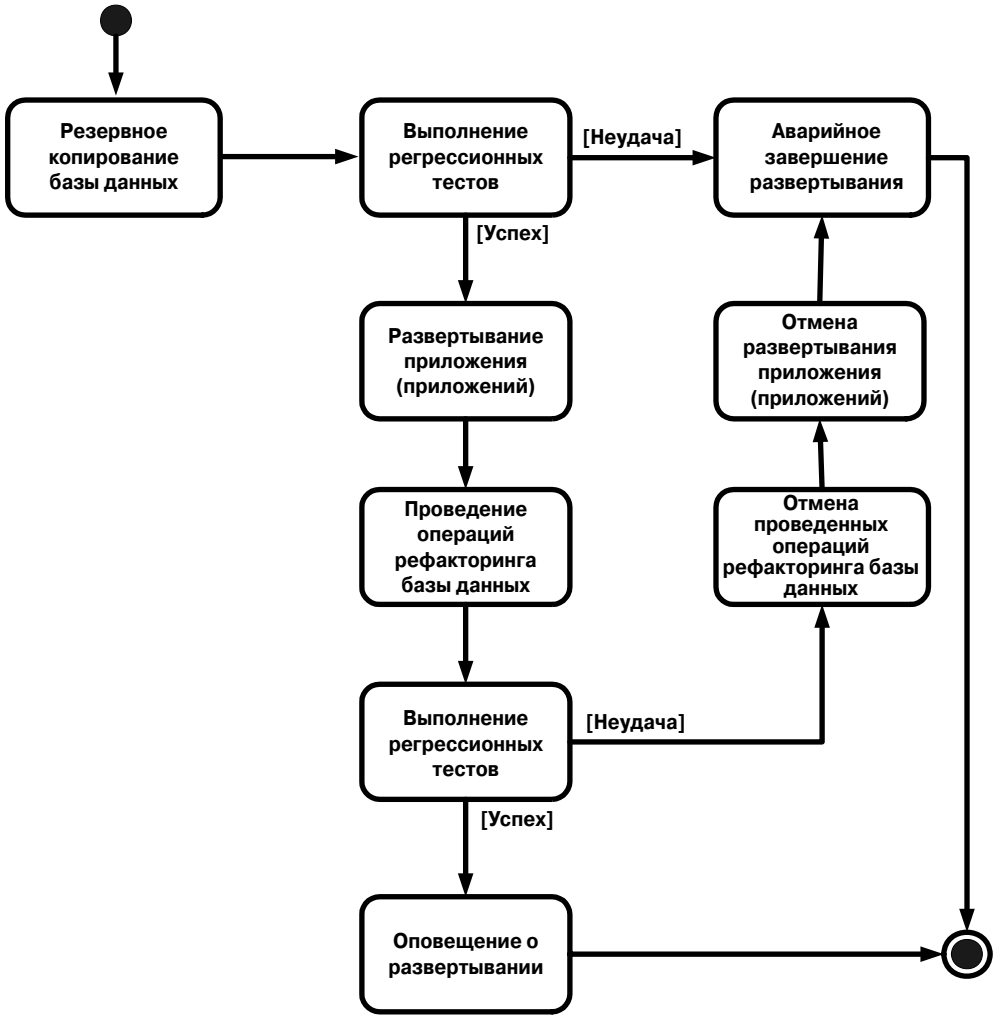


Рис. 4.4. Процесс развертывания

5. **Выполнение принятых в настоящее время регрессионных тестов.** После завершения развертывания приложения (приложений) и операций рефакторинга базы данных необходимо вызвать на выполнение текущую версию тестового набора для проверки того, что развернутая система (системы) действительно успешно работает в производственной среде. Еще раз отметим, что должна быть исключена возможность появления побочных эффектов при проведении тестов.
6. **В случае необходимости отмена внесенных изменений.** Если регрессионные тесты будут свидетельствовать о возникновении серьезных нарушений в работе, то необходимо возвратиться к предыдущим версиям приложений и схемы базы данных, а также восстановить базу данных с помощью резервной копии,

полученной на шаге 1. Если применяемый вариант развертывания является слишком сложным, то может потребоваться провести его поэтапно, осуществляя тестирование по завершении каждого этапа. Задача осуществления подхода, предусматривающего поэтапное развертывание, является более сложной, но обладает тем преимуществом, что применение этого подхода исключает возможность неудачного завершения всей процедуры развертывания из-за возникновения ошибок на единственном этапе.

7. **Публикация результатов развертывания.** После завершения успешного развертывания систем необходимо сразу же сообщить об этом всем заинтересованным лицам. Но даже если приходится завершать развертывание аварийно, все равно необходимо сообщить о том, что произошло, и когда будет предпринята повторная попытка развертывания. Необходимо учитывать, что есть и другие заинтересованные лица, которые надеются на успешное развертывание одной или нескольких необходимых им систем, поэтому желают знать о том, как обстоят дела.

4.5. Удаление устаревшей схемы

Операцию рефакторинга базы данных нельзя считать полностью развернутой до тех пор, пока из производственной среды не будет удалена устаревшая схема. После окончания переходного периода необходимо удалить устаревшую схему и весь вспомогательный код (такой как триггеры), который применялся для синхронизации различных версий схемы. Но переходный период может занять несколько лет, поскольку именно такое продолжительное время иногда требуется для обновления всех программ, получающих доступ к базе данных; из этого следует, что должен быть заранее организован процесс управления этими изменениями. Как было описано в главе 3, наиболее простой подход состоит в том, чтобы просто определить конкретные даты (возможно, отстоящие друг от друга на один квартал), в которые может оканчиваться переходный период. Это означает, что необходимо не только комплектовать операции, направленные на усовершенствование схемы базы данных, чтобы обеспечить применение сразу всех этих операций, но и объединять все действия по удалению устаревшей схемы и проводить сразу все необходимые действия.

Следует еще раз подчеркнуть, что и в этом случае должно быть предусмотрено тщательное тестирование. Перед удалением устаревших частей схемы из производственной среды необходимо вначале удалить их из среды обеспечения качества и проведения тестирования на этапе подготовки производства, а затем полностью повторить тестирование, чтобы убедиться в том, что не возникли какие-либо нарушения в работе. После выполнения этих действий необходимо реализовать такие изменения на производстве, выполнить тестовый набор в производственной среде, а затем либо произвести отмену, либо продолжить эксплуатацию с учетом изменений.

4.6. Резюме

Иначе говоря, операции рефакторинга базы данных после их реализации должны быть развернуты также на производстве, поскольку в противном случае проведение этих операций не имеет смысла. А благодаря наличию отдельных специализированных вариантов среды появляется возможность безопасно реализовывать и проверять операции рефакторинга, подготавливая их к развертыванию. В результате того, что разрабатываемые версии системы регулярно развертываются в среде тестирования на этапе подготовки производства, появляется возможность усовершенствовать и проверить сценарии развертывания задолго до того, как потребуется применить эти сценарии в производственной среде. Безусловно, ничто не препятствует стремлению постоянно проводить разработку программного обеспечения, предназначенного для эксплуатации в производственной среде, и выпускать новые версии, иногда даже каждую неделю, но, как правило, передача таких версий на производство не происходит столь же часто. Вместо этого имеет смысл комплектовать операции рефакторинга базы данных и развертывать целый набор таких операций одновременно в течение заранее определенного подходящего интервала развертывания. Тем не менее конкретное приложение может оказаться не единственным подлежащим развертыванию на производстве в течение определенного подходящего интервала развертывания, поэтому для успешного перехода к эксплуатации новых версий программного обеспечения, возможно, потребуется координировать свою работу с другими группами разработчиков.

Глава 5

Операции рефакторинга базы данных

Мы сегодня знаем больше, чем вчера, но это не дает нам оснований считать, что в этом в основном наша заслуга, а не заслуга наших предшественников.

Рон Джеффрис

В настоящей главе подытожен определенный опыт проведения операций рефакторинга баз данных в реальных проектах и предложены некоторые перспективные стратегии, которые могут быть приняты на вооружение. Поэтому данная глава может во многом рассматриваться как результат подведения итогов, полученных на практике, которые, по мнению авторов, могут помочь в освоении общего подхода, предусматривающего проведение рефакторинга. Ниже перечислены основные сделанные при этом выводы.

- Чем меньше изменения, тем проще их применение.
- Должна быть предусмотрена однозначная идентификация отдельных операций рефакторинга.
- Крупные изменения следует реализовывать в виде многочисленных мелких изменений.
- Необходимо использовать таблицу конфигурации базы данных.
- В качестве средства синхронизации предпочтительными являются триггеры, а не представления или пакеты.
- Продолжительность совместной эксплуатации новых и заменяемых средств должна быть достаточно велика.
- Группа контроля над внесением изменений (Change Control Board — CCB) в базу данных должна руководствоваться как можно более простой стратегией.
- Процедуры согласования с другими группами разработчиков должны быть максимально упрощены.

- Средства доступа к базе данных необходимо оформить в виде отдельных компонентов.
- Должна быть предусмотрена возможность легко настраивать среду базы данных.
- Необходимо исключить дублирование кода SQL.
- Информационные активы базы данных должны стать предметом контроля над внесением изменений.
- Должна учитываться необходимость перераспределения обязанностей в самой организации.

5.1. Преимущественное использование небольших изменений

Как правило, возникает соблазн применить к базе данных сразу несколько изменений. Например, почему бы не перенести за один прием столбец из одной таблицы в другую, переименовать его и перейти к использованию в нем стандартного типа данных? В этом стремлении нет ничего плохого, если не считать того, что на практике обнаруживается, что проведение подобных преобразований происходит сложнее, поэтому является более рискованным по сравнению с поэтапным выполнением отдельных указанных операций. Если в результате осуществления небольшого изменения будет обнаружено какое-либо нарушение в работе, то весьма велика вероятность того, что причину этого нарушения можно будет легко найти.

Снижение степени риска при реализации проекта в связи с применением небольших изменений

Подход, предусматривающий поэтапное осуществление небольших изменений, является более безопасным. Чем крупнее изменение, тем выше вероятность того, что его осуществление приведет к появлению каких-либо дефектов, и тем сложнее задача обнаружения любых дефектов, которые действительно внесены.

5.2. Однозначная идентификация отдельных операций рефакторинга

Чаще всего в ходе разработки проекта программного обеспечения приходится применять к схеме базы данных сотни операций рефакторинга и (или) преобразований. Многие из этих операций рефакторинга основаны на результатах других операций рефакторинга; например, может быть предусмотрено переименование столбца, а затем, по истечении нескольких недель, выполнен его перенос в другую таблицу, но при этом необходимо обеспечить, чтобы эти операции рефакторинга были применены в правильном порядке. Для этого должен быть предусмотрен какой-то конкретный способ обозначения всех операций рефакторинга, а также обозначения любых зависимостей между ними. В табл. 5.1 проводится сравнение и противопоставление трех основных стратегий решения указанной задачи. Стратегии, представленные в табл. 5.1, основаны на том предположении, что разработка осуществляется в среде с единственной базой данных и с одним приложением.

Таблица 5.1. Стратегии идентификации версий

Подход	Преимущества	Недостатки
Номер сборки. Номер сборки приложения, как правило, целое число, которое присваивается применяемым инструментальным средством сборки программного обеспечения (например, `CruiseControl`) после каждой компиляции приложения и успешного выполнения всех модульных тестов вслед за внесением изменений (даже если это изменение представляет собой операцию рефакторинга базы данных)	Простая стратегия. Операции рефакторинга могут рассматриваться как последовательная очередь изменений, применяемых в порядке, определяемом номером сборки. Версия базы данных непосредственно связана с версией приложения	Предполагается, что инструментальное средство рефакторинга баз данных объединено с инструментальным средством сборки приложений или что каждая операция рефакторинга реализована в виде одного или нескольких сценариев, выполняемых под контролем средств управления конфигурациями. Многие номера сборок относятся к версиям приложений, внедрение которых не связано с внесением изменений в базу данных. Поэтому идентификаторы версий применительно к базе данных не будут строго последовательными. (Например, изменения могут быть внесены в базу данных в связи с внедрением сборок приложений с номерами 1, 7, 11, 12..., а не 1, 2, 3, 4, ...) Задача управления усложняется, если одновременно разрабатывается несколько приложений, в которых используется одна и та же база данных, поскольку каждой группе разработчиков придется присваивать версиям своих приложений номера сборки, принадлежащие к одному и тому же ряду целых чисел
Отметка даты/времени. Операции рефакторинга присваивается текущее значение даты/времени	Простая стратегия. Управление операциями рефакторинга организуется в виде последовательной очереди	Если для реализации операций рефакторинга применяется подход на основе сценариев, то может стать затруднительным использование отметки даты/времени в качестве имени файла. Должна быть предусмотрена стратегия согласования операций рефакторинга с соответствующими им сборками приложений
Уникальный идентификатор. Операции рефакторинга присваивается уникальный идентификатор, такой как GUID или последовательно наращиваемое значение	Возможность применения существующих стратегий для выработки уникальных значений. (Например, может использоваться генератор GUID.)	Применение идентификаторов GUID в качестве имен файлов является затруднительным. В результате назначения операциям рефакторинга идентификаторов GUID не исключается необходимость обозначить последовательность их применения. Должна быть предусмотрена стратегия согласования операций рефакторинга с соответствующими им сборками приложений

Если деятельность по внесению изменений осуществляется в среде с несколькими приложениями, в которой могут возникать такие ситуации, что несколько групп разработчиков проектов применяют операции рефакторинга к одной и той же схеме базы данных, необходимо также найти способ, позволяющий определить, какой группой разработчиков была выполнена та или иная операция рефакторинга. Наиболее простой путь решения указанной задачи состоит в том, что каждой группе должен быть присвоен уникальный идентификатор, а в дальнейшем это значение должно включаться в состав идентификатора операции рефакторинга. Поэтому при использовании стратегии, предусматривающей определение номера сборки, группа 1 может осуществлять, допустим, операции рефакторинга с идентификаторами 1-7, 1-12, 1-15 и т.д., а группа 7 — операции рефакторинга с идентификаторами 7-3, 7-7, 7-13 и т.д.

Опыт авторов показал, что стратегия, в которой используются номера сборок, является наиболее эффективной, если за работу с базой данных отвечает лишь одна группа. Но если изменения в одну и ту же базу данных могут вносить несколько групп, то наиболее эффективным становится подход с применением отметок даты/времени, поскольку он позволяет легко определить по значению даты/времени, в каком порядке были проведены операции рефакторинга. Если же применяется подход на основе номеров сборок, то невозможно, например, определить, какая операция рефакторинга была выполнена первой, 1-7 или 7-7.

Но недостаточно просто предусмотреть последовательное применение операций рефакторинга. Один из авторов данной книги работал в организации, в которой разрешение на внесение изменений в одну и ту же схему базы данных было предоставлено четырем отдельным группам разработчиков. Кроме того, в организации было два администратора базы данных (DataBase Administrator — DBA), назовем их Федор и Борис, которые должны были обеспечить тесную координацию работы этих групп. Безусловно, администраторы базы данных прилагали значительные усилия по координации, причем чаще всего им удавалось добиться в этом успеха, но иногда возникали ошибки. Однажды Федор применил к определенному столбцу операцию рефакторинга “Применение стандартных кодовых обозначений” (с. 188), а через несколько дней Борис применил ту же операцию рефакторинга по просьбе другой группы разработчиков, но ввел при этом другой набор “стандартных” значений. Как оказалось, введенный Борисом набор “стандартных” значений был правильным, но вся эта ситуация обнаружилась лишь после того, как две группы разработчиков перенесли предусмотренные ими изменения в среду тестирования на этапе подготовки производства и фактически вступили в конфликт друг с другом. Из этого следует вывод, что в среде с несколькими группами разработчиков должна быть реализована определенная стратегия координации. (Несколько таких стратегий рассматриваются ниже в данной главе.)

5.3. Реализация крупных изменений в виде нескольких небольших изменений

Крупные изменения в базе данных, такие как реализация общей стратегии перехода к применению суррогатных ключей во всех таблицах или внедрение единообразной стратегии именования во всей базе данных, должны осуществляться в виде ряда небольших опе-

раций рефакторинга. Эта стратегия должна соответствовать шуточной рекомендации — "Как съесть слона? Разделив его на мелкие кусочки".

Рассмотрим задачу разбиения существующей таблицы на две. Безусловно, предусмотрена отдельная операция рефакторинга, называемая "Разбиение таблицы" (с. 177), но реальность такова, что на практике для решения этой задачи необходимо применить не одну, а несколько операций рефакторинга. Например, может потребоваться применить преобразование "Введение новой таблицы" (с. 323), чтобы добавить новую таблицу, несколько раз осуществить операцию рефакторинга "Перемещение столбца" (с. 139) (по одному разу для каждого столбца), чтобы переместить все столбцы, а также, в случае необходимости, провести операцию рефакторинга "Введение индекса" (с. 271), чтобы ввести первичный ключ в новую таблицу. Для осуществления каждой из операций рефакторинга "Перемещение столбца" необходимо несколько раз провести преобразование "Введение нового столбца" (с. 321) и преобразование "Перемещение данных" (с. 219). При выполнении этих действий может быть обнаружено, что требуется применить одну или несколько операций рефакторинга качества данных (подробнее об этом — в главе 7, "Операции рефакторинга качества данных"), чтобы повысить качество исходных данных в отдельных столбцах.

5.4. Применение таблицы конфигурации базы данных

В главе 3 обсуждалась необходимость идентификации текущей версии схемы базы данных для обеспечения возможности обновления этой схемы должным образом. Обозначения версий схемы должны соответствовать принятой стратегии рефакторинга базы данных; например, если для идентификации операций рефакторинга используется стратегия на основе отметок даты/времени, то необходимо также связать текущую версию схемы с отметкой даты/времени. Для этого проще всего предусмотреть таблицу, в которой будет сопровождаться эта информация. В следующем примере кода показано, как создать таблицу `DatabaseConfiguration` с одной строкой и с одним столбцом, которая позволяет реализовать стратегию определения номера сборки:

```
CREATE TABLE DatabaseConfiguration
 (SchemaVersion NUMBER NOT NULL);

INSERT INTO DatabaseConfiguration
 ( SchemaVersion ) VALUES (0);
```

Эта таблица обновляется, и в нее записывается новое значение идентификатора операции рефакторинга базы данных после применения каждой операции рефакторинга к базе данных. Например, после применения к схеме операции рефакторинга с номером 17 значение `DatabaseConfiguration.SchemaVersion` должно быть обновлено с указанием номера 17, как показано в следующем коде:

```
UPDATE DatabaseConfiguration
 SET SchemaVersion = 17;
```

5.5. Преимущественное применение для синхронизации триггеров, а не представлений или пакетов

В главе 2 было указано, что при обеспечении доступа нескольких приложений к одной и той же схеме базы данных часто требуется переходный период, в течение которого на производстве существуют одновременно две схемы, и исходная, и новая. Должен быть предусмотрен способ обеспечения того, чтобы приложения имели доступ к согласованным данным, независимо от того, какая версия схемы в них используется. В табл. 5.2 проводится сравнение и противопоставление трех основных стратегий, которые могут использоваться для обеспечения синхронизации данных. Опыт авторов показывает, что в подавляющем большинстве ситуаций наилучшим вариантом являются триггеры. Мы два-три раза использовали представления, а подход на основе пакетов применяли очень редко. Во всех примерах, приведенных в данной книге, предполагается, что синхронизация осуществляется на основе триггеров.

Таблица 5.2. Стратегии синхронизации схемы

Стратегия	Преимущества	Недостатки
Триггер. Применяется один или несколько триггеров, которые вносят соответствующие обновления в другую версию схемы	Обновление происходит в реальном времени	Потенциальная опасность появления узкого места производительности. Потенциальная опасность возникновения циклов. Потенциальная опасность возникновения взаимоблокировок. Увеличение объема дублирующихся данных. (Одни и те же данные хранятся и в исходной, и в новой схеме.)
Представления. Вводится представление (представления), соответствующее исходной таблице (таблицам); см. описание операции рефакторинга “Инкапсуляция таблицы в представление” (с. 266), которая позволяет обновлять обе схемы, и исходную, и новую, должным образом	Обновление происходит в реальном времени. Отсутствует необходимость перемещения физических данных из одной таблицы/столбца в другую таблицу/столбец	Обновляемые представления не поддерживаются в некоторых базах данных; еще один вариант состоит в том, что база данных может не поддерживать соединения, в которых участвуют обновляемые представления. Дополнительные сложности возникают при вводе в действие и в конечном итоге удалении представления (представлений)
Пакетные обновления. Пакетное задание, в котором данные обрабатываются и обновляются должным образом, регулярно вызывается на выполнение (например, один раз в сутки)	Отрицательное влияние на производительность, связанное с осуществлением синхронизации данных, сводится к минимуму благодаря тому, что обновление производится во время непиковых нагрузок	Исключительно высокая опасность возникновения проблем нарушения ссылочной целостности. Приходится отслеживать предыдущие версии данных, чтобы иметь возможность определить, какие изменения были внесены в конкретную строку.

Окончание табл. 5.2

Стратегия	Преимущества	Недостатки
		Если в период между проведениями пакетных обновлений было внесено несколько изменений (например, выполнено обновление данных, содержащихся в обеих схемах, и в исходной, и в новой), то становится сложно определить, какое изменение (изменения) должно быть принято.
		Увеличение объема дублирующихся данных. (Одни и те же данные хранятся и в исходной, и в новой схеме.)

5.6. Применение достаточно продолжительного переходного периода

Администратор базы данных должен отвести для осуществления операции рефакторинга переходный период с обоснованной продолжительностью, которая является достаточной для всех других групп разработчиков приложения. Авторы пришли к выводу, что наиболее простым является такой подход, который предусматривает совместную выработку общего решения о применении одинакового переходного периода для различных категорий операций рефакторинга, а в дальнейшем — неуклонное применение такого переходного периода. Например, предположим, что для операций рефакторинга структуры может быть принят двухлетний переходный период, а для операций рефакторинга архитектуры — переходный период с продолжительностью три года. Основным недостатком этого подхода является то, что он требует принятия наиболее продолжительных переходных периодов из всех возможных, даже если рефакторингу подвергается схема, к которой получает доступ небольшое число часто модернизируемых приложений. Остроту указанной проблемы можно смягчить, своевременно удаляя из производственных баз данных те части схемы, которые больше не требуются, даже несмотря на то, что переходный период мог еще не закончиться, согласовывая решение о применении более короткого переходного периода с группой контроля над внесением изменений в базу данных или проводя непосредственное согласование с другими группами разработчиков.

5.7. Упрощение стратегии группы контроля над внесением изменений в базу данных

Скотт некогда работал в одной компании, в которой была группа контроля над внесением изменений в базу данных, состоящая из штатных администраторов базы данных, имеющих полное представление об информационных активах предприятия. Представители этой группы контроля над внесением изменений в базу данных встречались один раз в неделю и на совещания группы администраторы баз данных, занимающиеся разработкой проектов, приносили списки предлагаемых изменений, которые соответствующая группа разработчиков хотела бы внести в существующие производственные источники

данных. (Группам разработчиков никто не препятствовал вносить изменения в источники данных, которые еще не применялись на производстве.) Группа контроля над внесением изменений в базу данных определяла, следует ли разрешить проведение предложенных изменений, и в случае положительного ответа принималось решение о том, каким должен быть переходный период. Преимущество этого подхода состояло в том, что всегда оставалось ощущение неусыпного наблюдения со стороны группы контроля над внесением изменений в базу данных. (Безусловно, такая стратегия стала бы полностью непродуктивной, если бы представители группы разработчиков решили вносить изменения без консультаций с кем-либо.) С другой стороны, указанная организация работы имеет один недостаток, связанный с тем, что оперативность действий всех групп снижается. Даже если решение по проведению изменений будет принято в течение нескольких часов или двух-трех дней, все равно группе разработчиков приходится откладывать мероприятия по переходу на новую схему.

5.8. Упрощение процедуры согласования с другими группами

Третья стратегия определения переходного периода состоит в том, что отдельные операции рефакторинга должны согласовываться с владельцами всех систем, которые могут быть затронуты той или иной операцией рефакторинга базы данных. Такие согласования могут проводиться по отношению к каждой отдельной операции рефакторинга или по отношению к целому пакету операций во время проведения регулярного совещания по согласованию изменений в схеме базы данных. Преимуществом такого подхода является то, что он позволяет сообщить о потенциальных изменениях всем заинтересованным лицам и тем самым способствует более точному определению переходного периода (при условии, что все присутствующие на совещании смогут точно спрогнозировать предполагаемое время развертывания изменений, в которых они должны участвовать). А основной недостаток состоит в том, что этот подход может быть связан с большими затратами времени и со значительными затруднениями. Авторы еще ни разу не были свидетелями попыток его применения на практике. Но тем, кто действительно попытается его применить, мы рекомендуем до предела упростить все процедуры, связанные с внесением изменений.

5.9. Инкапсуляция средств доступа к базе данных

В главе 2 было указано, что по мере дальнейшей инкапсуляции средств доступа к базе данных задача проведения рефакторинга базы данных все больше упрощается. По крайней мере, даже если приложение включает жестко закодированные программные конструкции SQL, нужно стремиться к тому, чтобы код SQL был сосредоточен в одном месте, что позволило бы легко находить и обновлять определенные фрагменты этого кода по мере необходимости. Более высокая степень инкапсуляции может предусматривать применение для реализации программных средств SQL единообразного способа, например, подготовку операций сохранения, удаления, выборки и поиска, `save()`, `delete()`, `retrieve()` и `find()`, для каждого класса деловых сущностей. Еще один вариант может предусматривать реализацию объектов доступа к данным (Data Access Object — DAO), т.е. классов, которые реализуют программные средства доступа к данным отдельно от

классов деловых сущностей. Например, могут быть предусмотрены отдельные классы `CustomerDAO` и `AccountDAO` соответственно, для деловых сущностей `Customer` и `Account`. Еще более перспективный подход состоит в том, что можно полностью отказаться от применения заранее подготовленного кода SQL и формировать процедуры доступа к базе данных на основе отображений метаданных [**4**]. Общий вывод из сказанного таков, что по мере упрощения задачи проведения рефакторинга схемы базы данных становится проще находить, а затем обновлять средства доступа к базе данных.

5.10. Возможность легко настраивать среду базы данных

В ходе реализации жизненного цикла проекта почти невозможно избежать того, что одни люди будут подключаться к осуществлению проекта, а другие — выходить из состава участников разработки проекта. Как показано на рис. 5.1, все группы разработчиков проекта должны иметь возможность создавать экземпляры базы данных, при этом зачастую развертывая разные версии схемы на компьютерах различных классов, в соответствии с описанием, приведенным в главе 4. Наиболее эффективный способ достижения этой цели состоит в подготовке сценария инсталляции, который позволил бы применить код DDL для первоначального создания схемы базы данных, а также всех соответствующих сценариев внесения изменений, с последующим выполнением набора регрессионных тестов в целях проверки успешного проведения инсталляции.

Рис. 5.1. Специализированные варианты среды

5.11. Предотвращение дублирования кода SQL

Одним из замечательных свойств языка SQL является то, что на нем можно довольно легко написать требуемый прикладной код. К сожалению, такое удобное свойство языка SQL становится причиной того, что, как обнаружили авторы, код SQL часто дублируется по всему приложению и даже по всей базе данных, неоднократно повторяясь в представлениях, хранимых процедурах и триггерах. Но чем больший объем кода SQL находится в

вашем распоряжении, тем сложнее становится задача проведения рефакторинга схемы базы данных, поскольку увеличивается вероятность обнаружить намного больше кода, связанного с тем объектом, который должен быть подвергнут рефакторингу. Поэтому лучше всего применять код SQL в виде отдельных пакетов или классов, предусматривать формирование кода SQL на основе метаданных или сохранять код SQL в файлах XML, доступ к которым осуществляется на этапе прогона.

5.12. Перевод информационных активов базы данных под контроль процедур управления изменениями

В главе 1 было показано, насколько важно перевести все информационные активы базы данных, такие как модели данных и сценарии базы данных, под контроль процедур управления изменениями. Обоим авторам этой книги доводилось участвовать в работе групп проектировщиков, в которых это не было сделано администраторами базы данных, а иногда даже разработчиками. Как и следовало ожидать, в подобных группах часто приходится прилагать значительные усилия, пытаясь определить надлежащую версию модели данных или найти требуемый сценарий внесения изменений, когда наступает время развертывать создаваемые приложения в среде тестирования на этапе подготовки производства, а иногда даже на производстве. Необходимо обеспечить эффективное управление не только важными информационными активами проекта, но и информационными активами самой базы данных. Авторы пришли к выводу, что наиболее продуктивный подход состоит в обеспечении совместного хранения информационных активов базы данных в том же репозитарии, где хранятся компоненты приложения, поскольку это позволяет определить, кем были внесены изменения, а также отменить неудачные изменения в случае необходимости.

5.13. Учет необходимости перераспределения обязанностей в самой организации

Не исключено, что внедрение эволюционных методов сопровождения базы данных, в частности подхода, основанного на проведении рефакторинга базы данных, приведет к существенным изменениям в самой организации. Как отметили Маннс и Райзинг [**28**] в своей книге *Fearless Change*, при попытке добиться надлежащего осуществления подобных усовершенствованных методов, вероятно, придется столкнуться с определенными политическими трениями в самой организации. Безусловно, многие специалисты в области информационной технологии предпочитают игнорировать политические аспекты своей деятельности, наивно полагая, что эти аспекты их не коснутся, но тем самым они подвергаются большой опасности. Методы, описанные в данной книге, являются работоспособными; мы знаем об этом, поскольку проверили их на практике. Нам также известно, что многие специалисты в области обработки данных, которые являются приверженцами давно сложившихся подходов, сопротивляются внедрению новых методов, и в этом они правы, поскольку в результате вся организация их работы в будущем изменится коренным образом. Любому, кто захочет внедрить в своей организации подход, основанный на проведении рефакторинга базы данных, по-видимому, придется вступить в “политические игры”.

5.14. Резюме

Рефакторинг баз данных — это относительно новая методика разработки, но уже нередко встречаются такие специалисты, которые могут гордиться значительными успехами, достигнутым в этом направлении. В настоящей главе авторы представили некоторые результаты накопленного ими опыта и предложили несколько новых стратегий, которыми многие могут с успехом воспользоваться.

Сетевые ресурсы

Авторы сопровождают список рассылки *Agile Databases* на узле *Yahoo Groups*. Эта группа новостей имеет URL-адрес `groups.yahoo.com/group/agileDatabases/`. Приглашаем читателей принять участие в обсуждениях рассматриваемых тем и поделиться своим опытом.

Кроме того, мы сопровождаем узлы `www.databaserefactoring.com` и `www.agiledata.org`, на которых поддерживаются актуальные списки операций рефакторинга базы данных. Новые операции рефакторинга добавляются по мере их обнаружения.

Глава 6

Операции рефакторинга структуры

Операции рефакторинга структуры, в соответствии с их названием, позволяют изменить структуру таблиц в схеме базы данных. К операциям рефакторинга структуры относятся следующие.

- Операция рефакторинга “Удаление столбца”.
- Операция рефакторинга “Удаление таблицы”.
- Операция рефакторинга “Удаление представления”.
- Операция рефакторинга “Введение вычисляемого столбца”.
- Операция рефакторинга “Введение суррогатного ключа”.
- Операция рефакторинга “Слияние столбцов”.
- Операция рефакторинга “Слияние таблиц”.
- Операция рефакторинга “Перемещение столбца”.
- Операция рефакторинга “Переименование столбца”.
- Операция рефакторинга “Переименование таблицы”.
- Операция рефакторинга “Переименование представления”.
- Операция рефакторинга “Замена данных типа LOB таблицей”.
- Операция рефакторинга “Замена столбца”.
- Операция рефакторинга “Замена связи “один ко многим” ассоциативной таблицей”.
- Операция рефакторинга “Замена суррогатного ключа естественным ключом”.
- Операция рефакторинга “Разбиение столбца”.
- Операция рефакторинга “Разбиение таблицы”.

Проблемы, часто возникающие при реализации операций рефакторинга структуры

При реализации операций рефакторинга структуры приходится учитывать несколько часто возникающих проблем, связанных с обновлением схемы базы данных, включая перечисленные ниже.

1. **Предотвращение триггерных циклов.** Триггер должен быть реализован так, чтобы не возникали циклы; например, если должно произойти обновление в одном из исходных столбцов, `Table.NewColumn1..N`, это обновление не должно привести к запуску триггера для такого же обновления исходных столбцов и т.д. Приведенный ниже код может служить примером того, как обеспечить синхронизацию значений двух столбцов.

```
CREATE OR REPLACE TRIGGER SynchronizeFirstName
 BEFORE INSERT OR UPDATE
 ON Customer
 REFERENCING OLD AS OLD NEW AS NEW
 FOR EACH ROW
 DECLARE
 BEGIN
 IF INSERTING THEN
         IF :NEW.FirstName IS NULL THEN
          :NEW.FirstName := :NEW.FName;
         END IF;
         IF :NEW.Fname IS NULL THEN
          :NEW.FName := :NEW.FirstName;
         END IF;
        END IF;
        IF UPDATING THEN
          IF NOT(:NEW.FirstName=:OLD.FirstName) THEN
           :NEW.FName:=:NEW.FirstName;
          END IF;
          IF NOT(:NEW.FName=:OLD.FName) THEN
           :NEW.FirstName:=:NEW.FName;
          END IF;
         END IF;
        END;
       /
```

2. **Исправление нарушенных представлений.** Представления связаны с другими частями базы данных, поэтому после применения любой операции рефакторинга структуры может оказаться, что непреднамеренно нарушено функционирование представления. Определения представлений обычно содержат ссылки на определения других представлений и таблиц. Например, предположим, что представление `VCustomerBalance` определено путем соединения таблиц `Customer` и `Account` для получения с помощью столбца `CustomerNumber` итогового баланса по всем счетам каждого отдельного клиента. Если столбец `Customer.CustomerNumber` будет переименован, это приведет к нарушению работы представления.

3. **Исправление нарушенных триггеров.** Триггеры связаны с определениями таблиц, поэтому такие структурные изменения, как переименование или перемещение столбца, могут привести к нарушению работы одного из триггеров. Например, предположим, что для проверки достоверности данных, хранящихся в определенном столбце, применяется триггер вставки; в таком случае после внесения изменений в определение этого столбца работа триггера потенциально может быть нарушена. Следующий код показывает, как найти нарушенные триггеры в СУБД Oracle; рекомендуем включить такую процедуру в свой тестовый набор. Но для выявления всех дефектов бизнес-логики потребуются также другие тесты.

```
SELECT Object_Name, Status
          FROM User_Objects
WHERE Object_Type='TRIGGER'
AND Status='INVALID';
```

5. **Исправление нарушенных хранимых процедур.** Хранимые процедуры вызывают другие хранимые процедуры, а также обеспечивают доступ к таблицам, представлениям и столбцам. Поэтому любые операции рефакторинга структуры связаны с риском нарушения работы существующей хранимой процедуры. Приведенный ниже код показывает, как найти нарушенные хранимые процедуры в СУБД Oracle; рекомендуем включить такую процедуру в свой тестовый набор. Но для выявления всех дефектов бизнес-логики потребуются также другие тесты.

```
SELECT Object_Name, Status
          FROM User_Objects
WHERE Object_Type='PROCEDURE'
AND Status='INVALID';
```

6. **Исправление нарушенных таблиц.** Таблицы косвенно связаны со столбцами других таблиц в соответствии с соглашениями об именовании. Например, в случае переименования столбца `Customer.CustomerNumber` может потребоваться аналогичным образом переименовать столбцы `Account.CustomerNumber` и `Policy.CustomerNumber`. Следующий код позволяет найти все таблицы с именами столбцов, содержащими текст `CUSTOMERNUMBER`, в СУБД Oracle:

```
SELECT Table_Name,Column_Name
          FROM User_Tab_Columns
WHERE Column_Name LIKE '%CUSTOMERNUMBER%';
```

7. **Определение переходного периода.** Для осуществления операций рефакторинга структуры требуется переходный период, если такие операции применяются в среде с несколькими приложениями. Необходимо назначить одни и те же даты внесения изменений не только в исходную схему, подвергающуюся рефакторингу, но и в любые столбцы и триггеры. При выборе этой даты внесения изменений необходимо учитывать время, которое потребуется для обновления внешних программ, получающих доступ к затронутой изменениями части базы данных.

Операция рефакторинга “Удаление столбца”

Эта операция рефакторинга предусматривает удаление столбцов из существующей таблицы (рис. 6.1).

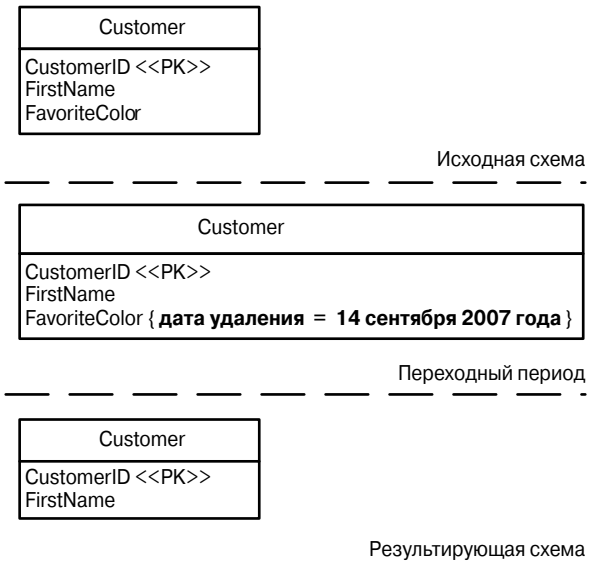


Рис. 6.1. Удаление столбца `Customer.FavoriteColor`

Обоснование

Основная причина применения операции “Удаление столбца” состоит в том, чтобы подвергнуть рефакторингу проект таблицы базы данных или учесть результат операций рефакторинга внешних приложений, после проведения которых столбец больше не используется. Операция “Удаление столбца” часто применяется как один из шагов операции “Перемещение столбца” (с. 139) рефакторинга базы данных, связанных с удалением столбца из исходной таблицы. Еще одна причина использования этой операции состоит в том, что иногда обнаруживается, что некоторые из столбцов фактически не применяются. Обычно лучше сразу удалить эти столбцы, прежде чем кто-либо начнет использовать их по ошибке.

Потенциальные преимущества и недостатки

Удаляемый столбец может содержать ценные данные; в таком случае, возможно, потребуется сохранение этих данных. Чтобы сохранить данные, можно переместить их в какую-то другую таблицу с помощью операции “Перемещение данных” (с. 219). Выполнение операции “Удаление столбца” на таблицах, содержащих много строк, может привести к тому, что удаление столбца потребует много времени, поэтому таблица станет недоступной для обновления.

Процедура обновления схемы

Чтобы обновить схему для удаления столбца, необходимо выполнить указанные ниже действия.

1. **Выбрать стратегию удаления.** Может оказаться так, что некоторые программные продукты баз данных не позволяют удалять столбцы. В таком случае необходимо создать временную таблицу, переместить в нее все данные, удалить исходную таблицу, затем воссоздать ее без удаляемого столбца, переместить данные из временной таблицы, а затем удалить временную таблицу. Если же база данных предоставляет способ удаления столбцов, то достаточно только удалить столбец с помощью опции `DROP COLUMN` команды `ALTER TABLE`.
2. **Удалить столбец.** Иногда необходимо убедиться в том, что операция “Удаление столбца” будет выполнена за приемлемое время, если количество данных в таблице достаточно велико. Чтобы свести к минимуму нарушение нормального хода работы, планируйте физическое удаление столбца на то время, когда таблица используется меньше всего. Еще один подход состоит в том, что столбец базы данных может быть отмечен как неиспользуемый; такую операцию можно осуществить с помощью опции `SET UNUSED` команды `ALTER TABLE`. Команда с опцией `SET UNUSED` выполняется намного быстрее по сравнению с удалением, поэтому нарушение нормального хода работы сводится к минимуму. В таком случае неиспользуемые столбцы можно удалить в течение запланированного времени простоя. Если используется данный вариант, то столбец в базе данных физически не удаляется, а становится скрытым от всех.
3. **Переопределить внешние ключи.** Допустим, что удаляемый столбец `FavoriteColor` входит в состав первичного ключа; это означает, что необходимо также удалить соответствующие столбцы из других таблиц, которые используют его как внешний ключ (или часть ключа) к таблице `Customer`. На всех прочих таблицах необходимо воссоздать ограничения внешнего ключа. В подобной ситуации для упрощения операции рефакторинга может оказаться целесообразным вначале применить такую операцию рефакторинга, как “Введение суррогатного ключа” (с. 123) или “Замена суррогатного ключа естественным ключом” (с. 168), а затем воспользоваться операцией “Удаление столбца”.

   Еще одним вариантом, который может применяться вместо физического удаления столбца, является сокрытие существования этого столбца. Для этого необходимо ввести представление таблицы, которое не ссылается на столбец `FavoriteColor`, применив операцию рефакторинга “Инкапсуляция таблицы в представление” (с. 266).

   При этом достаточно лишь связать со столбцом `Customer.FavoriteColor` комментарий для указания на то, что этот стобец будет вскоре удален; комментарий остается в силе в течение переходного периода. По истечении переходного периода столбец удаляется из таблицы `Customer` с помощью команды `ALTER TABLE`, как показано ниже.

```
COMMENT ON Customer.FavoriteColor 'Drop date = September 14 2007';

--14 сентября 2007 года
ALTER TABLE Customer DROP COLUMN FavoriteColor;
```

Если используется опция `SET UNUSED`, то можно применить следующую команду, для того чтобы сделать столбец `Customer.FavoriteColor` неиспользуемым; это позволяет в действительности не удалять его из таблицы `Customer` физически, но сделать недоступным и невидимым для всех клиентов:

```
ALTER TABLE Customer SET UNUSED FavoriteColor;
```

Процедура переноса данных

Может оказаться так, что для поддержки операции удаления столбца из таблицы необходимо сохранить существующие данные, или, возможно, операцию "Удаление столбца" требуется запланировать таким образом, чтобы ее выполнение не привело к снижению производительности (поскольку на то время, когда происходит удаление столбца из таблицы, возможность модификации данных в таблице исключается). Основная проблема в данном случае состоит в том, что, прежде чем удалять столбец, необходимо сохранить данные. Кроме того, если намечено удаление существующего столбца из таблицы, которая находится в эксплуатации, то, возможно, представители предприятия потребуют сохранить существующие данные "на всякий случай", поскольку эти данные снова могут потребоваться в какой-то момент в будущем. Самый простой подход состоит в том, чтобы создать временную таблицу с первичным ключом исходной таблицы и столбцом, подлежащим удалению, а затем переместить соответствующие данные в эту новую временную таблицу. Могут быть также выбраны другие методы сохранения данных, такие как архивирование данных во внешних файлах.

Ниже приведен код, который иллюстрирует этапы удаления столбца `Customer.FavoriteColor`. Чтобы сохранить эти данные, необходимо создать временную таблицу `CustomerFavoriteColor`, которая включает первичный ключ из таблицы `Customer` и столбец `FavoriteColor`.

```
CREATE TABLE CustomerFavoriteColor
  AS SELECT CustomerID, FavoriteColor FROM Customer;
```

Процедура обновления программ доступа

Необходимо выявить, а затем обновить все внешние программы, которые ссылаются на столбец `Customer.FavoriteColor`. Ниже перечислены проблемы, которые следует проанализировать в связи с этим.

1. **Подвергнуть код рефакторингу в целях использования альтернативных источников данных.** Может оказаться так, что некоторые внешние программы включают код, в котором все еще используются данные, содержащиеся в настоящее время в столбце `Customer.FavoriteColor`. Если дело обстоит таким образом, то необходимо найти альтернативные источники данных и переопределить код в целях их применения; в противном случае придется отказаться от операции рефакторинга.

2. **Уменьшить количество данных, подвергаемых выборке с помощью операторов `SELECT`.** Некоторые внешние программы могут включать запросы, которые считывают данные, а затем игнорируют значения, полученные в результате выборки.
3. **Подвергнуть рефакторингу операторы вставки и обновления базы данных.** Некоторые внешние программы могут включать код, который помещает “фиктивные значения” в столбец, подлежащий удалению, при вставке новых данных; этот код должен быть удален. Еще один вариант состоит в том, что программы могут включать код, не предусматривающий перезапись значений в стобце `Favorite-Color` во время вставки или обновления данных в базе данных. В других случаях может оказаться, что применяется оператор `SELECT * FROM Customer`, при выполнении которого в приложении предусматривается получение определенного количества столбцов, и столбцы из результирующего набора берутся с использованием позиционной ссылки. После удаления указанного столбца работа такого прикладного кода, по-видимому, нарушится, поскольку результирующий набор оператора `SELECT` теперь будет возвращать на один столбец меньше. Вообще говоря, в приложениях не рекомендуется использовать форму оператора `SELECT *` для выборки данных из любой таблицы. Очевидно, что фактически в данном случае проблема состоит в том, что в приложении используются позиционные ссылки, а это указывает на необходимость применения операции рефакторинга для устранения и этого недостатка.

Ниже приведен код, который показывает, как следует удалять ссылку на столбец `FavoriteColor`.

```
// Код до рефакторинга
public Customer findByCustomerID(Long customerID) {
  stmt = DB.prepare("SELECT CustomerId, FirstName, "+
    "FavoriteColor FROM Customer WHERE CustomerId = ?");
  stmt.setLong(1, customerID.longValue());
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerId(rs.getLong("CustomerId"));
    customer.setFirstName(rs.getString("FirstName"));
    customer.setFavoriteColor(rs.getString("FavoriteColor"));
  }
  return customer;
}

public void insert(long customerId,String firstName, String
favoriteColor) {
  stmt = DB.prepare("INSERT into customer" +
    "(Customerid, FirstName, FavoriteColor) " +
    "values (?, ?, ?)");
  stmt.setLong(1, customerId);
  stmt.setString(2, firstName);
  stmt.setString(3, favoriteColor);
  stmt.execute();
}
```

```
public void update(long customerId, String firstName, String color) {
  stmt = DB.prepare("UPDATE Customer "+
  "SET FirstName = ?, FavoriteColor=? " +
  "WHERE Customerid = ?");
  stmt.setString(1, firstName);
  stmt.setString(2, color);
  stmt.setLong(3, customerId);
  stmt.executeUpdate();
}

// Код после рефакторинга

public Customer findByCustomerID(Long customerID) {
  stmt = DB.prepare("SELECT CustomerId, FirstName " +
    "FROM Customer WHERE CustomerId = ?");
  stmt.setLong(1, customerID.longValue());
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerId(rs.getLong("CustomerId"));
    customer.setFirstName(rs.getString("FirstName"));
  }
  return customer;
}

public void insert(long customerId,String firstName) {
  stmt = DB.prepare("INSERT into customer" +
    "(Customerid, FirstName) " +
    "values (?, ?)");
  stmt.setLong(1, customerId);
  stmt.setString(2, firstName);
  stmt.execute();
}

public void update(long customerId, String firstName, String color) {
  stmt = DB.prepare("UPDATE Customer "+
  "SET FirstName = ? " +
  "WHERE Customerid = ?");
  stmt.setString(1, firstName);
  stmt.setLong(2, customerId);
  stmt.executeUpdate();
}
```

Операция рефакторинга “Удаление таблицы”

Удаление существующей таблицы из базы данных (рис. 6.2).

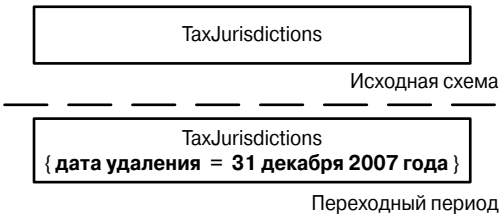


Рис. 6.2. Удаление таблицы `TaxJurisdictions`

Обоснование

Операция “Удаление таблицы” применяется, если таблица больше не требуется и (или) не используется. Такая ситуация возникает после того, как таблица заменяется другим аналогичным источником данных, например другой таблицей или представлением, или же после того, как этот конкретный источник данных становится больше не нужным.

Потенциальные преимущества и недостатки

В результате удаления таблицы происходит также удаление из базы данных определенных данных, поэтому может потребоваться сохранить некоторые или все эти данные. Если дело обстоит таким образом, то требуемые данные необходимо сохранить в другом источнике данных, особенно если осуществляется нормализация проекта базы данных и обнаруживается, что некоторые из данных существуют в другой таблице (таблицах). Кроме того, вместо таблицы может применяться представление или запрос к источнику данных. Но в таком случае исключается возможность записи данных с помощью того же представления или запроса к источнику данных.

Процедура обновления схемы

Для осуществления операции “Удаление таблицы” необходимо прежде всего решить проблемы целостности данных. Например, если на таблицу `TaxJurisdictions` ссылаются какие-то другие таблицы, то необходимо либо удалить ограничение внешнего ключа, либо направить ограничение внешнего ключа на другую таблицу. На рис. 6.2 приведен пример того, как обеспечить удаление таблицы `TaxJurisdictions`; для этого достаточно просто обозначить эту таблицу как запрещенную для применения, а затем удалить ее после наступления даты перехода. Ниже приведен код DDL, предназначенный для удаления таблицы.

```
-- Дата удаления - 14 июня 2007 года
DROP TABLE TaxJurisdictions;
```

Кроме того, как показано ниже, можно также выбрать вариант, предусматривающий только переименование таблицы. Некоторые программные продукты баз данных при выполнении такой операции автоматически переназначают все ссылки с таблицы TaxJurisdictions на таблицу TaxJurisdictionsRemoved. Может также потребоваться удалить подобные ограничения ссылочной целостности с помощью операции "Уничтожение ограничения внешнего ключа" (с. 238), поскольку нежелательно, чтобы на удаляемую таблицу распространялись какие-либо ограничения ссылочной целостности:

```
-- Дата переименования - 14 июня 2007 года
ALTER TABLE TaxJurisdictions RENAME TO TaxJurisdictionsRemoved;
```

Процедура переноса данных

Единственная проблема переноса данных, связанная с осуществлением этой операции рефакторинга, обусловлена тем, что в принципе может потребоваться архивировать существующие данные, чтобы их можно было в дальнейшем восстановить в случае необходимости. Это можно сделать с помощью команды CREATE TABLE AS SELECT. В следующем коде приведены операторы DDL, которые при желании могут использоваться для сохранения данных, находящихся в таблице TaxJurisdictions:

```
-- Скопировать данные перед удалением
CREATE TABLE TaxJurisdictionsRemoved AS
          SELECT * FROM TaxJurisdictions;

-- Дата удаления - 14 июня 2007 года
DROP TABLE TaxJurisdictions;
```

Процедура обновления программ доступа

Рефакторингу должны быть подвергнуты все внешние программы, ссылающиеся на таблицу TaxJurisdictions, чтобы предоставить им доступ к альтернативному источнику (источникам) данных, заменяющему таблицу TaxJurisdictions. Если альтернативные источники данных отсутствуют, а данные, хранящиеся в таблице, все еще требуются, то не следует удалять таблицу до тех пор, пока не появится альтернативный источник (источники) данных.

Операция рефакторинга "Удаление представления"

Эта операция обеспечивает удаление существующего представления (рис. 6.3).

Обоснование

Операция "Удаление представления" применяется, если представление больше не требуется и (или) не используется. Такая ситуация возникает, если представление заменяется иным аналогичным источником данных, таким как другое представление или таблица, или просто отпадает необходимость в использовании запроса, лежащего в основе данного конкретного представления.

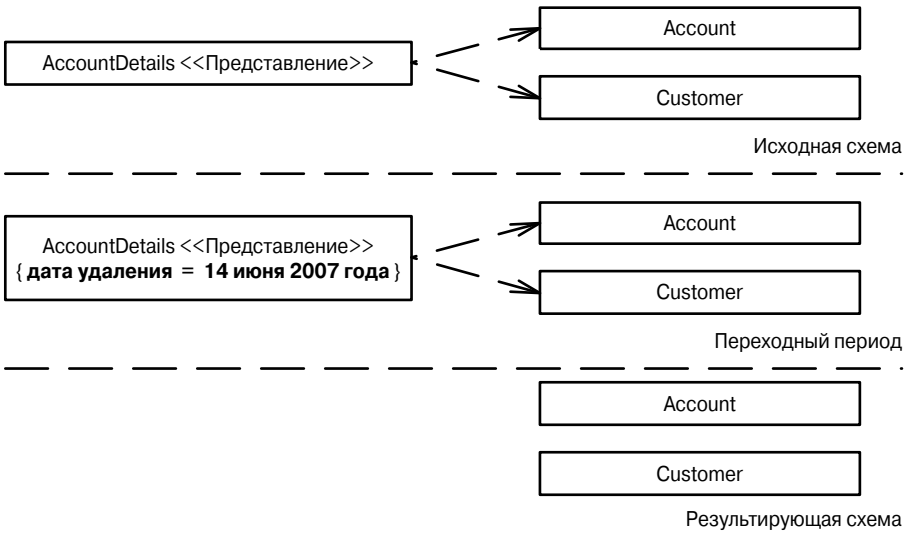


Рис. 6.3. Удаление представления `AccountDetails`

Потенциальные преимущества и недостатки

Удаление представления не приводит к удалению данных из базы данных, но влечет за собой то, что представление становится неприменимым для внешних программ, которые к нему обращались. Представления часто используются при получении данных для отчетов. Если такие данные все еще требуются, то представление так или иначе потребуется заменить другим источником данных, т.е. представлением или таблицей, либо запросом непосредственно к исходным данным. В идеальном случае новый способ доступа к данным должен обеспечивать такую же или более высокую производительность, что и удаляемое представление. Представления используются также для реализации управления защитой доступа (Security Access Control — SAC) к значениям данных, хранящимся в базе данных. Если дело обстоит таким образом, то должна быть предварительно реализована и развернута новая стратегия SAC применительно к таблицам, доступ к которым предоставлялся с помощью представления. Стратегия обеспечения безопасности на основе представлений часто является наиболее простым подходом, который может совместно использоваться для поддержки многих приложений, но она не является столь же гибкой, как программная стратегия SAC [**4**].

Процедура обновления схемы

Для удаления представления, показанного на рис. 6.3, необходимо применить команду `DROP VIEW` к представлению `AccountDetails` по истечении переходного периода. Приведенный ниже код, обеспечивающий удаление представления `AccountDetails`, является очень простым; достаточно лишь обозначить представление как устаревшее, а затем удалить его после наступления даты перехода.

```
-- Дата удаления - 14 июня 2007 года
DROP VIEW AccountDetails;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных данные, предназначенные для переноса, отсутствуют.

Процедура обновления программ доступа

Необходимо выявить, а затем обновить все внешние программы, которые ссылаются на представление `AccountDetails`. Может также потребоваться подвергнуть рефакторингу код SQL, в котором прежде использовалось представление `AccountDetails`, чтобы теперь доступ осуществлялся непосредственно к данным из исходных таблиц. Аналогичным образом, требуют обновления все метаданные, используемые для выработки кода SQL с помощью представления `AccountDetails`. Следующий код показывает, как откорректировать прикладной код, чтобы вместо представления использовались данные из базовых таблиц:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT * " +
  "FROM AccountDetails "+
  "WHERE CustomerId = ?");
  stmt.setLong(1,customer.getCustomerID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
"SELECT * " +
"FROM Customer, Account " +
"WHERE " +
"  Customer.CustomerId = Account.CustomerId " +
"  AND Customer.CustomerId = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга “Введение вычисляемого столбца”

Эта операция предназначена для введения нового столбца, создаваемого с помощью вычислений, в которых применяются данные из одной или нескольких таблиц. (На рис. 6.4 показаны две таблицы, но количество таких таблиц может быть произвольным.)

Обоснование

Основная причина, по которой может потребоваться применение операции “Введение вычисляемого столбца”, состоит в повышении производительности приложения путем предоставления заранее вычисленных значений для какого-то конкретного свойства, производного по отношению к другим данным. Например, может потребоваться ввести вычисляемый столбец, который указывает уровень кредитного риска

(допустим, “образцовый клиент”, “допустимый риск”, “недопустимый риск”) применительно к какому-то клиенту на основании предыдущей хронологии расчетов клиента с конкретной фирмой.

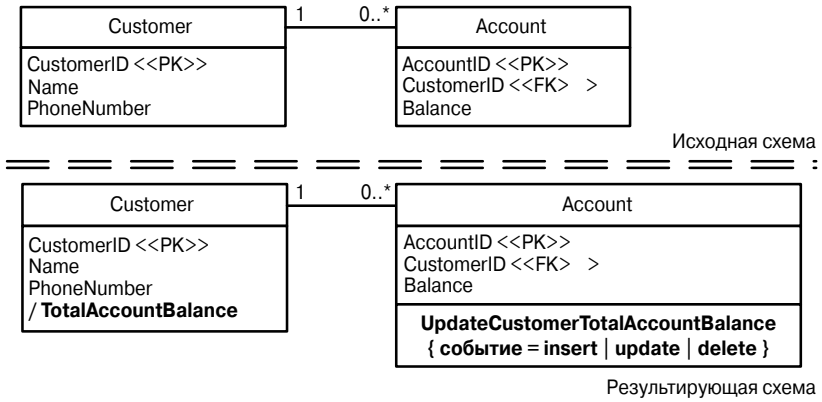


Рис. 6.4. Введение вычисляемого столбца `Customer.TotalAccountBalance`

Потенциальные преимущества и недостатки

Вычисляемый столбец может выйти из синхронизации с фактическими значениями данных, особенно если для обновления в нем значений требуются внешние приложения. Авторы предлагают ввести в действие какой-то механизм, позволяющий автоматически обновлять значения в вычисляемом столбце, например, обычное пакетное задание или триггеры, которые определены на исходных данных.

Процедура обновления схемы

Применение операции “Введение вычисляемого столбца” может оказаться сложным из-за наличия зависимостей в данных, а также в связи с необходимостью поддерживать синхронизацию значений в вычисляемом столбце со значениями данных, на которых он основан. В частности, необходимо выполнить описанные ниже действия.

1. **Определить стратегию синхронизации.** Основными вариантами являются пакетные задания, прикладные обновления или триггеры базы данных. Пакетное задание может использоваться, если не требуется обновление значений в режиме реального времени; в противном случае необходимо применить одну из двух прочих стратегий. Если ответственность за выполнение соответствующих обновлений возлагается на какое-то приложение (приложения), то возникает риск, связанный с тем, что в различных приложениях обновление может осуществляться по-разному. Подход, основанный на применении триггера, по-видимому, является наиболее безопасным из двух стратегий обновления в реальном времени, поскольку соответствующий программный код должен быть реализован только единожды в базе данных. В ситуации, показанной на рис. 6.4, предполагается использование триггеров.

2. **Определить способ вычисления значения.** Необходимо выявить исходные данные и определить способ их применения для вычисления значения `TotalAccountBalance`.
3. **Определить таблицу, содержащую необходимый столбец.** Необходимо уточнить, какая таблица должна включить столбец `TotalAccountBalance`. Для этого достаточно выяснить, какую деловую сущность лучше всего описывает этот вычисляемый столбец. Например, индикатор кредитного риска клиента является в наибольшей степени применимым к сущности `Customer`.
4. **Добавить новый столбец.** Необходимо добавить столбец `Customer.TotalAccountBalance`, показанный на рис. 6.4, с помощью преобразования "Введение нового столбца" (с. 321).
5. **Реализовать стратегию обновления.** Для этого необходимо реализовать и проверить стратегию, выбранную на шаге 1.

Ниже приведен код, который показывает, как ввести дополнительно столбец `Customer.TotalAccountBalance` и триггер `UpdateCustomerTotalAccountBalance`, вызываемый на выполнение при внесении каждого изменения в данные таблицы `Account`.

```
-- Создать новый столбец TotalAccountBalance
ALTER TABLE Customer ADD TotalAccountBalance NUMBER;

-- Создать триггер для обеспечения синхронизации данных

CREATE OR REPLACE TRIGGER
UpdateCustomerTotalAccountBalance
BEFORE UPDATE OR INSERT OR DELETE
ON Account
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
NewBalanceToUpdate NUMBER:=0;
CustomerIdToUpdate NUMBER;
BEGIN
CustomerIdToUpdate := :NEW.CustomerID;
IF UPDATING THEN
  NewBalanceToUpdate := :NEW.Balance-:OLD.Balance;
END IF;
IF INSERTING THEN
  NewBalanceToUpdate := :NEW.Balance;
END IF;
IF DELETING THEN
  NewBalanceToUpdate := -1*:OLD.Balance;
  CustomerIdToUpdate := :OLD.CustomerID;
END IF;
UPDATE Customer SET TotalAccountBalance =
  TotalAccountBalance + NewBalanceToUpdate
  WHERE Customerid = CustomerIdToUpdate;
END;
/
```

Процедура переноса данных

Как таковой, перенос данных не требуется, хотя должно быть предусмотрено заполнение значений столбца `Customer.TotalAccountBalance` с помощью вычислений. Как правило, такая операция осуществляется один раз с использованием команды `UPDATE` языка SQL или может быть выполнена в пакетном задании, включающем один или несколько сценариев. Ниже приведен код, который показывает, как задать начальное значение в столбце `Customer.TotalAccountBalance`.

```
UPDATE Customer SET
  TotalAccountBalance =
    (SELECT SUM(balance) FROM Account
      WHERE Account.CustomerId = Customer.CustomerId)
```

Процедура обновления программ доступа

После ввода в действие вычисляемого столбца необходимо выявить все те участки кода во внешних приложениях, где используются соответствующие вычисления, а затем переопределить код таким образом, чтобы он работал со значениями `TotalAccountBalance`. Таким образом, необходимо заменить существующий код, выполняющий вычисления, кодом доступа к значениям `TotalAccountBalance`. При этом может быть обнаружено, что в различных приложениях вычисления выполняются по-разному, либо по ошибке, либо по какой-то другой причине, поэтому потребуется согласовать правильный алгоритм со всеми пользователями базы данных. В следующем коде показано, что в одном из приложений вычисление итогового баланса осуществляется путем обработки в цикле всех счетов клиента. А в версии, обозначенной как “код после рефакторинга”, просто считывается значение в память после выборки данных из объекта клиента в базе данных:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT SUM(Account.Balance) Balance FROM Customer, Account " +
    "WHERE Customer.CustomerID = Account.CustomerID "+
    "AND Customer.CustomerID=?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
return rs.getBigDecimal("Balance"));

// Код после рефакторинга
return customer.getBalance();
```

Операция рефакторинга “Введение суррогатного ключа”

Эта операция предусматривает замену существующего естественного ключа суррогатным ключом (рис. 6.5). Данная операция рефакторинга является противоположной по отношению к операции “Замена суррогатного ключа естественным ключом” (с. 168).

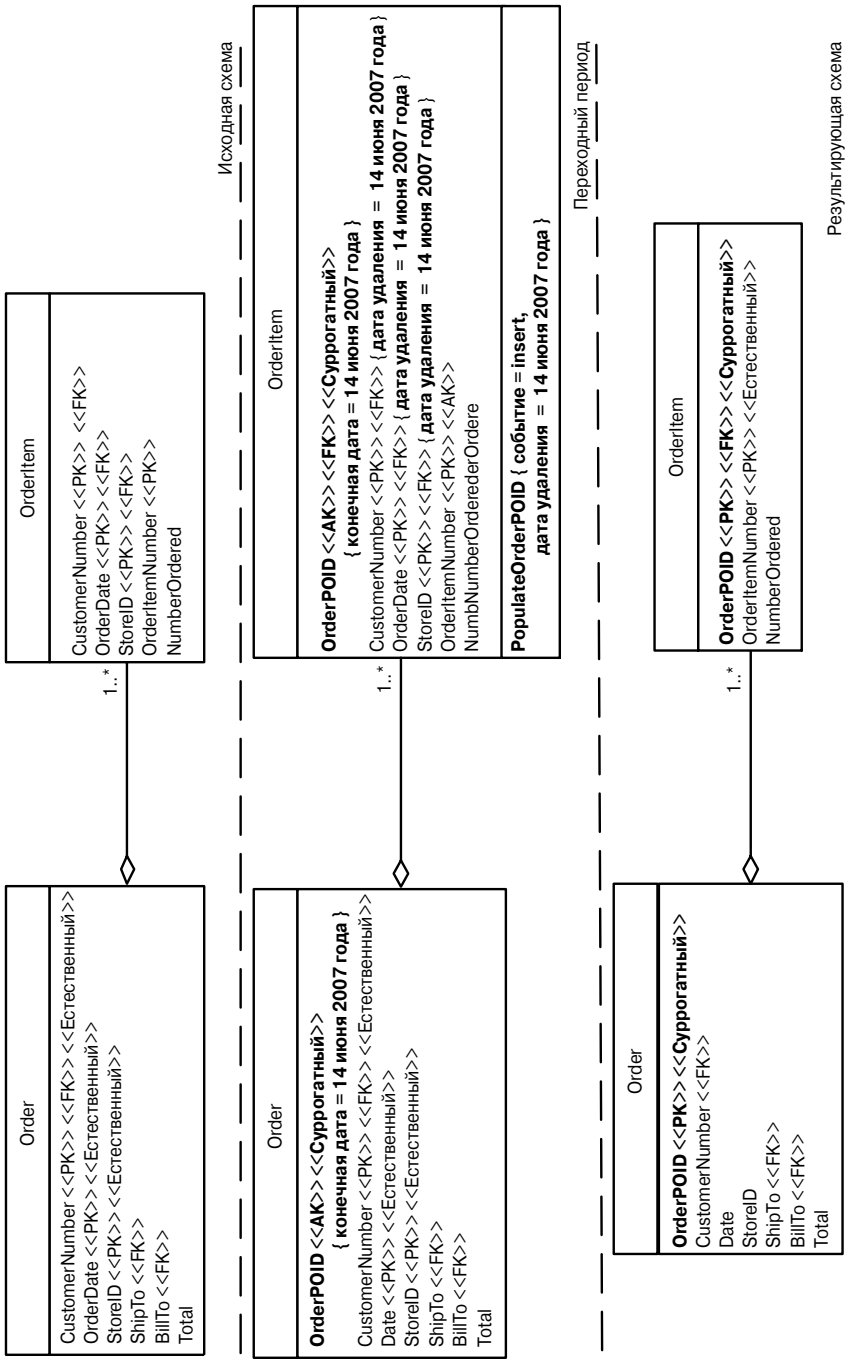


Рис. 6.5. Ввод в действие суррогатного ключа `Order.OrderPOID`

Обоснование

Введение суррогатного ключа в таблицу может потребоваться по нескольким причинам, описанным ниже.

- **Уменьшение степени связности.** Это — основная причина, которая обусловлена необходимостью сократить степень связности между схемой таблицы и доменом данных делового предприятия. Если существует вероятность того, что часть естественного ключа в дальнейшем изменится, например, увеличатся размеры представления номеров деталей, хранящихся в таблице запасов, или произойдет переход от одного типа данных к другому (допустим, от числового к алфавитно-цифровому), то решение об использовании такого естественного ключа в качестве первичного является очень опасным.
- **Повышение единообразия.** Может потребоваться применить операцию рефакторинга “Осуществление стратегии консолидированных ключей” (с. 197), которая потенциально позволяет повысить производительность и сократить сложность кода.
- **Повышение производительности базы данных.** В связи с применением крупного составного естественного ключа производительность базы данных может снизиться. (Некоторые проектировщики баз данных стремятся всеми силами избежать применения любых ключей, состоящих из нескольких столбцов.) После того как крупный составной первичный ключ заменяется суррогатным первичным ключом с единственным столбцом, база данных приобретает способность обновлять индекс.

Потенциальные преимущества и недостатки

Многие специалисты в области баз данных предпочитают естественные ключи. Споры в сообществе проектировщиков баз данных между сторонниками и противниками суррогатных и естественных ключей иногда напоминают религиозные войны, но реальность такова, что оба типа ключей имеют свою область применения. Даже если в таблице имеется суррогатный первичный ключ, все равно могут потребоваться альтернативные естественные ключи для обеспечения поиска. Конечные пользователи не могут использовать суррогатные ключи для поиска, поскольку они не имеют никакого делового смысла и обычно реализуются в виде коллекции бессодержательных символов или чисел. В связи с этим для пользователей все равно остается необходимость идентифицировать данные с помощью естественных идентификаторов. Например, предположим, что в таблице `InventoryItem` имеются суррогатный первичный ключ `InventoryItemPOID`, POID (Persistent Object Identifier — постоянный идентификатор объекта) и естественный альтернативный ключ `InventoryID`. Однозначная идентификация отдельных элементов в системе осуществляется с помощью суррогатного ключа, а пользователи идентифицируют эти элементы, применяя естественный ключ. С другой стороны, важность применения суррогатных ключей состоит в том, что они позволяют упростить стратегию сопровождения ключей в базе данных и уменьшить степень связности между схемой базы данных и доменом данных делового предприятия.

Еще одним возможным источником нарушений в работе может служить то, что реализован суррогатный ключ, который в действительности не требуется. Многие специалисты теряют чувство меры, когда дело касается реализации ключей, и часто пытаются применить одну и ту же стратегию во всей создаваемой схеме. Например, в Соединенных

Штатах отдельные штаты обозначаются уникальным двухбуквенным кодом штата (в частности, код CA обозначает Калифорнию). Гарантируется, что этот код штата является уникальным в пределах Соединенных Штатов и Канады; например, кодом канадской провинции Онтарио является ON, и не может существовать ни одного американского штата с тем же кодом. Штаты и провинции — это довольно стабильные сущности, и к тому же остается не используемым все еще достаточно большое количество кодов (до настоящего времени применялось только 65 из 676 возможных комбинаций), а соответствующие правительственные органы вряд ли изменят такую стратегию кодирования, поскольку опасаются неразберихи, которая может возникнуть из-за этого в их собственных системах. Иначе говоря, имеет ли смысл вводить суррогатный ключ на поисковой таблице, в которой перечислены все штаты и провинции? По-видимому, нет.

Кроме того, если первоначальный ключ OriginalKey используется в качестве внешнего ключа в других таблицах, то может потребоваться применить операцию "Осуществление стратегии консолидированных ключей" (с. 197) и внести аналогичные изменения в эти и другие таблицы. Следует отметить, что иногда выполнение подобной работы не оправдывается, поэтому необходимо еще раз взвесить, стоит ли применять эту операцию рефакторинга.

Процедура обновления схемы

Применение операции "Введение суррогатного ключа" может стать затруднительным из-за связности, которой потенциально может характеризоваться исходный ключ (в рассматриваемом примере — комбинация CustomerNumber, OrderDate и StoreID). Кроме того, этот ключ представляет собой первичный ключ таблицы, поэтому не исключена возможность того, что он образует также внешний ключ (или его часть), который указывает на таблицу Order из других таблиц. Необходимо выполнить описанные ниже действия.

1. **Ввести новый ключевой столбец.** Добавить столбец к целевой таблице с помощью команды ADD COLUMN языка SQL. На рис. 6.5 таковым является столбец OrderPOID. Этот столбец необходимо будет заполнить уникальными значениями.
2. **Добавить новый индекс.** Для таблицы Order должен быть введен новый индекс на основе OrderPOID.
3. **Обозначить исходный столбец как устаревший.** Первоначальные ключевые столбцы должны быть отмечены как предназначенные для понижения статуса до столбцов альтернативного ключа или до неключевых столбцов в конце переходного периода, в зависимости от обстоятельств. В рассматриваемом примере в данный момент не происходит удаление столбца из таблицы Order, но такие столбцы уже не рассматриваются как относящиеся к первичному ключу. Тем не менее эти столбцы будут удалены из таблицы OrderItem.
4. **Обновить и, возможно, добавить триггеры ссылочной целостности (Referential Integrity — RI).** Все существующие триггеры, предназначенные для поддержки отношений ссылочной целостности между таблицами, должны быть обновлены, для того чтобы они могли работать с соответствующими новыми значениями ключа в других таблицах. Должны быть также введены триггеры для заполнения

значениями столбцов внешнего ключа на время переходного периода, поскольку может оказаться, что приложения еще не обновлены с учетом необходимости выполнения этих действий.

На рис. 6.5 показано, как ввести в таблицу `Order` суррогатный ключ `OrderPOID`. Необходимо также рекурсивно применить операцию “Введение суррогатного ключа” к таблице `OrderItem` (см. рис. 6.5), чтобы иметь возможность использовать новый ключевой столбец. Но это действие не является обязательным. Безусловно, в таблице `OrderItem` все еще может применяться существующий составной ключ, состоящий из столбцов `CustomerNumber`, `OrderDate` и `StoreID`, но для единообразия было решено подвергнуть рефакторингу также и эту таблицу.

Ниже приведен код SQL, который позволяет ввести и обеспечить первоначальное заполнение столбцов `OrderPOID` в таблицах `OrderItem` и `Order`. В этом коде уникальные значения для столбца `OrderPOID` формируются посредством вызова хранимой процедуры `GenerateUniqueID`, которая реализует алгоритм `HIGH-LOW` [**4**]. В этом коде вводится также соответствующий индекс, необходимый для сопровождения столбца `OrderPOID` как ключевого для таблицы `Order`.

```
-- Добавить новый суррогатный ключ к таблице Order
ALTER TABLE Order ADD OrderPOID NUMBER;

-- Добавить новый внешний суррогатный ключ к таблице OrderItem
ALTER TABLE OrderItem ADD OrderPOID NUMBER;

--Присвоить значения столбцу суррогатного ключа в таблице Order
UPDATE Order SET OrderPOID =
getOrderPOIDFromOrder(CustomerNumber,OrderDate,StoreID);

-- Распространить действие ключа ForeignKey на таблицу OrderItem
UPDATE OrderItem SET OrderPOID =
  (SELECT OrderPOID FROM Order
    WHERE CustomerNumber = Order.CustomerNumber
      AND OrderDate = Order.OrderDate
      AND StoreID=Order.StoreID);

CREATE INDEX OrderOrderPOIDIndex ON Order (OrderPOID);
```

Для поддержки нового ключа необходимо дополнительно ввести триггер `Populate-OrderPOID`, который вызывается при выполнении каждой операции вставки в таблицу `OrderItem`. Этот триггер получает значения из столбца `Order.OrderPOID`, как показано в следующем коде SQL:

```
CREATE OR REPLACE TRIGGER PopulateOrderPOID
BEFORE INSERT
ON OrderItem
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
IF :NEW.OrderPOID IS NULL THEN
```

```
  :NEW.OrderPOID :=
getOrderPOIDFromOrder(CustomerNumber,OrderDate,StoreID);
END IF;
IF :NEW.OrderPOID IS NOT NULL THEN
  IF :NEW.CustomerNumber IS NULL
    OR :NEW.OrderDate IS NULL
    OR :NEW.StoreID IS NULL
  THEN
    :NEW.CustomerNumber
      := getCustomerNumberFromOrder(OrderPOID);
      :NEW.OrderDate
:= getOrderDateFromOrder(OrderPOID);
  :NEW.StoreID
:= getStoreIDFromOrder(OrderPOID);
  END IF;
END IF;
END;
/

-- 14 июня 2007 года
ALTER TABLE OrderItem DROP CONSTRAINT OrderItemToOrderForeignKey;

ALTER TABLE Order DROP CONSTRAINT OrderPrimaryKey;

ALTER TABLE Order MODIFY OrderPOID NOT NULL;

ALTER TABLE Order ADD CONSTRAINT OrderPrimaryKey
  PRIMARY KEY (OrderPOID);

ALTER TABLE OrderItem DROP CONSTRAINT OrderItemPrimaryKey;

ALTER TABLE OrderItem MODIFY OrderPOID NOT NULL;
ALTER TABLE OrderItem ADD CONSTRAINT OrderItemPrimaryKey
  PRIMARY KEY (OrderPOID, OrderItemNumber);

ALTER TABLE OrderItem ADD (CONSTRAINT OrderItemToOrderForeignKey
  FOREIGN KEY (OrderPOID) REFERENCES Order;

CREATE UNIQUE INDEX OrderNaturalKey ON Order
(CustomerNumber,OrderDate,StoreID);

DROP TRIGGER PopulateOrderPOID;
```

Процедура переноса данных

Необходимо выработать данные для заполнения столбца `Order.OrderPOID` и присвоить эти значения столбцам внешнего ключа в других таблицах.

Процедура обновления программ доступа

Для работы со столбцом `Order.OrderPOID` должны быть обновлены все внешние программы, ссылающиеся на первоначальные ключевые столбцы. В частности, может потребоваться переработка кода для выполнения описанных ниже действий.

1. **Присваивание ключевых значений новых типов.** Если для присваивания новых значений суррогатного ключа служит внешний прикладной код, а не сама база данных, то должны быть переопределены все внешние приложения, чтобы с их помощью присваивались значения столбцу `Order.OrderPOID`. При этом минимальным требованием является то, чтобы в каждой отдельной программе был реализован один и тот же алгоритм выполнения этой операции, но лучшая стратегия состоит в том, что должна быть реализована общая служба, которую вызывает каждое приложение.
2. **Организация выполнения соединений на основе нового ключа.** Может оказаться так, что во многих внешних программах доступа определены соединения, в которых участвует таблица `Order`, реализованные с помощью программных конструкций SQL или метаданных. Такие операции соединения должны быть подвергнуты рефакторингу, чтобы они могли работать со столбцом `Order.OrderPOID`.
3. **Осуществление выборки на основе нового ключа.** В некоторых внешних программах предусматривается прохождение по базе данных путем единовременного получения одной или нескольких строк; при этом выборка данных осуществляется на основе значений ключа. Такие операции выборки должны быть обновлены для обеспечения работы со столбцом `Order.OrderPOID`.

Ниже приведено Hibernate-отображение (`www.hibernate.org`), которое показывает, как ввести суррогатный ключ.

```
// Отображение до рефакторинга
<hibernate-mapping>
  <class name="Order" table="ORDER">
    <many-to-one name="customer"
    class="Customer" column="CUSTOMERNUMBER" />
    <property name="orderDate"/>
    <property name="storeID"/>
    <property name="shipTo"/>
    <property name="billTo"/>
    <property name="total"/>
  </class>
</hibernate-mapping>

// Отображение после рефакторинга
<hibernate-mapping>
  <class name="Order" table="ORDER">
    <id name="id" column="ORDERPOID">
      <generator class="OrderPOIDGenerator"/>
    </id>
    <many-to-one name="customer"
     class="Customer" column="CUSTOMERNUMBER" />
    <property name="orderDate"/>
    <property name="storeID"/>
    <property name="shipTo"/>
    <property name="billTo"/>
    <property name="total"/>
  </class>
</hibernate-mapping>
```

Операция рефакторинга “Слияние столбцов”

Эта операция обеспечивает слияние двух или нескольких столбцов, относящихся к одной таблице (рис. 6.6).

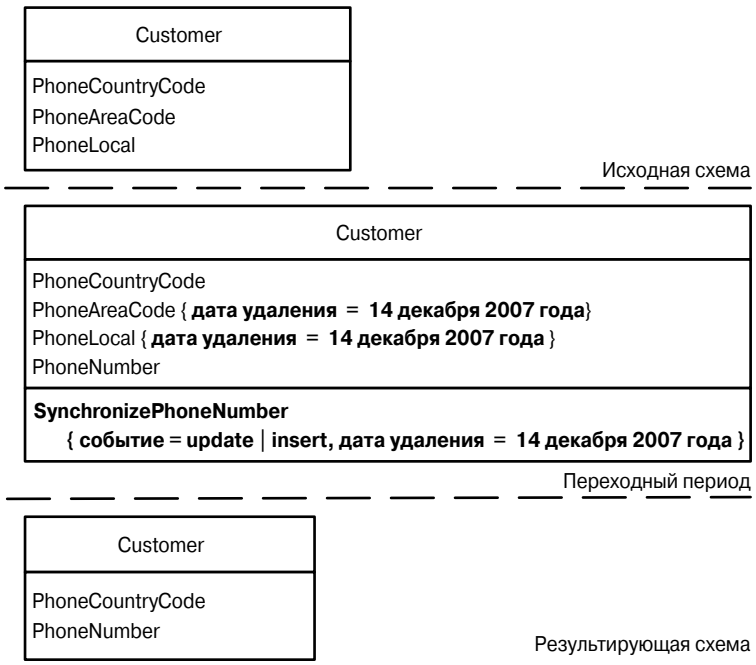


Рис. 6.6. Слияние столбцов в таблице `Customer`

Обоснование

Необходимость в применении операции “Слияние столбцов” может быть обусловлена несколькими причинами, в том числе указанными ниже.

- **Наличие идентичных столбцов.** Возможно, двое или несколько разработчиков ввели одинаковые столбцы без ведома друг друга; такая ситуация встречается часто, если разработчики относятся к разным коллективам или отсутствуют метаданные, описывающие схему таблицы. Например, предположим, что в таблице `FeeStructure` имеется 37 столбцов; два из них называются `CA_INIT` и `CheckingAccountOpeningFee`, и оба содержат данные об авансовом платеже, взимаемом банком при открытии текущего счета. Столбец `CheckingAccountOpeningFee` был добавлен, поскольку никто из разработчиков не смог со всей уверенностью ответить, для чего действительно предназначен столбец `CA_INIT`.

- **В таблице имеются столбцы, явившиеся результатом чрезмерно тщательного проектирования.** Первоначально столбцы были введены для обеспечения хранения информации в виде ее составляющих, но в ходе реальной эксплуатации было обнаружено, что фактически не требуется такая тонкая детализация, которую предполагалось использовать при создании проекта. Например, на рис. 6.6 показано, что таблица `Customer` включает столбцы `PhoneCountryCode`, `PhoneAreaCode` и `PhoneLocal`, которые представляют единственный номер телефона в виде кода страны, кода региона и локального номера телефона.
- **Со временем разные столбцы фактически приобрели одинаковое назначение.** В таблицу первоначально было введено несколько столбцов, но по истечении определенного времени способ использования одного или нескольких из этих столбцов изменился так, что теперь эти столбцы в действительности применяются для одной и той же цели. Например, предположим, что таблица `Customer` включает столбцы `PreferredCheckStyle` и `SelectedCheckStyle` (не показанные на рис. 6.6). Первый столбец использовался для регистрации данных о том, какой стиль оформления применялся для счетов, передаваемых клиенту, который выбрал себе товары к следующему сезону, а второй столбец был предназначен для хранения данных о стиле оформления счетов, передававшихся клиенту ранее. Применение обоих этих столбцов имело смысл двадцать лет тому назад, когда требовалось несколько месяцев для перехода к оформлению заказов с помощью счетов с новым форматом, а в настоящее время необходимые бланки счетов могут быть напечатаны за одну ночь, поэтому предприятие автоматически перешло к использованию такого варианта организации работы, при котором в обоих столбцах хранятся одинаковые значения.

Потенциальные преимущества и недостатки

Выполнение этой операции рефакторинга базы данных может привести к потере точности данных после слияния столбцов, имеющих большую степень детализации. Кроме того, если осуществляется слияние столбцов, которые (по вашему мнению) используются для одной и той же цели, то вы подвергаетесь риску столкнуться с тем, что эти столбцы фактически используются для различных назначений. (В таком случае вы обнаружите, что должны повторно ввести один или несколько исходных столбцов.) Таким образом, решение о слиянии столбцов следует принимать с учетом области применения данных, но для этого необходимо ознакомиться с мнениями тех, кто использует эти столбцы в своей работе.

Процедура обновления схемы

Для осуществления операции “Слияние столбцов” необходимо выполнить два действия. Вначале необходимо ввести новый столбец. Добавьте столбец в таблицу с помощью команды `ADD COLUMN` языка SQL. На рис. 6.6 таковым является столбец `Customer.PhoneNumber`. Этот шаг необязателен, поскольку может оказаться, что есть возможность использовать один из существующих столбцов, в который будут перенесены данные в результате слияния. Необходимо также ввести триггер синхронизации для обеспечения того, чтобы столбцы оставались синхронизированными друг с другом. Этот триггер должен вызываться при внесении любого изменения в столбцы.

На рис. 6.6 показан пример, в котором первоначально в таблице Customer номера телефонов клиентов хранились в трех отдельных столбцах: PhoneCountryCode, PhoneAreaCode и PhoneLocal. Со временем было обнаружено, что код страны требуется лишь для немногих приложений, поскольку подавляющая часть приложений эксплуатируется только в Северной Америке. В ходе эксплуатации был также сделан вывод, что в каждом приложении используются вместе и код города, и локальный номер телефона. Поэтому было решено оставить незатронутым столбец PhoneCountryCode, но объединить столбцы PhoneAreaCode и PhoneLocal в один столбец PhoneNumber, что соответствует фактической форме использования данных в приложениях (поскольку в приложениях данные PhoneAreaCode или PhoneLocal не применяются отдельно). Кроме того, было предусмотрено введение триггера SynchronizePhoneNumber, который позволил бы поддерживать значения во всех четырех столбцах синхронизированными.

Ниже приведен код SQL с операторами DDL, позволяющими ввести столбец PhoneNumber и в конечном итоге удалить два исходных столбца.

```
ALTER TABLE Customer ADD PhoneNumber NUMBER(12);
          COMMENT ON Customer.PhoneNumber 'Added as the
          result of merging Customer.PhoneAreaCode and
          Customer.PhoneLocal finaldate = December 14 2007';

          -- 14 декабря 2007 года
          ALTER TABLE Customer DROP COLUMN PhoneAreaCode;
          ALTER TABLE Customer DROP COLUMN PhoneLocal;
```

Процедура переноса данных

Необходимо преобразовать все данные из формата исходного столбца (столбцов) в формат столбца, создаваемого в результате слияния; в рассматриваемом примере речь идет о переносе данных из столбцов Customer.PhoneAreaCode и Customer.PhoneLocal в столбец Customer.PhoneNumber. В следующем коде SQL приведены операторы DML, позволяющие осуществить первоначальное заполнение столбца PhoneNumber, объединив в нем данные из столбцов PhoneAreaCode и PhoneLocal.

```
/* Одноразовый перенос данных из таблиц Customer.PhoneAreaCode и
Customer.PhoneLocal в таблицу Customer.PhoneNumber. Если оба столбца
активны, необходимо предусмотреть триггер, который обеспечивает
синхронизацию обоих столбцов */
          UPDATE Customer SET PhoneNumber =
  PhoneAreaCode*10000000 + PhoneLocal);
```

Процедура обновления программ доступа

Необходимо тщательно проанализировать программы доступа, а затем обновить их должным образом в течение переходного периода. Кроме этой очевидной доработки, необходимо перейти к применению столбца Customer.PhoneNumber, а не прежних необъединенных столбцов. После этого, возможно, придется удалить код, который применялся для объединения данных, находившихся в исходных столбцах. При этом речь может идти о том коде, в котором данные из существовавших ранее столбцов комбинировались в атрибуты данных, аналогичные тем, которые хранятся в столбце, полученном в

результате слияния. Этот код должен быть подвергнут рефакторингу и, возможно, полностью удален.

Кроме того, может также потребоваться обновить код проверки данных, чтобы он мог работать с данными, полученными в результате слияния. Часть существующего кода может применяться исключительно в связи с тем, что слияние столбцов еще не было выполнено. Например, если какое-то значение хранится в двух отдельных столбцах, то может потребоваться применение кода проверки, позволяющего следить за тем, чтобы значения в этих столбцах были согласованными. А после объединения столбцов, возможно, этот код больше не потребуется.

Ниже приведен код, состоящий из двух фрагментов, применяемых до и после слияния, которые показывают, как изменяется метод `getCustomerPhoneNumber()` после слияния столбцов `Customer.PhoneAreaCode` и `Customer.PhoneLocal`.

```
// Код до рефакторинга
public String getCustomerPhoneNumber(Customer customer) {
  String phoneNumber = customer.getCountryCode();
  phoneNumber.concat(phoneNumberDelimiter());
  phoneNumber.concat(customer.getPhoneAreaCode());
  phoneNumber.concat(customer.getPhoneLocal());
  return phoneNumber;
}

// Код после рефакторинга
public String getCustomerPhoneNumber(Customer customer) {
  String phoneNumber = customer.getCountryCode();
  phoneNumber.concat(phoneNumberDelimiter());
  phoneNumber.concat(customer.getPhoneNumber());
  return phoneNumber;
}
```

Операция рефакторинга “Слияние таблиц”

Эта операция позволяет выполнить слияние двух или нескольких таблиц в одну таблицу (рис. 6.7).

Обоснование

Применение операции “Слияние таблиц” может потребоваться по нескольким причинам, описанным ниже.

- **Некоторые таблицы стали результатом чрезмерно детализированного проектирования.** Исходные таблицы были введены для обеспечения того, чтобы информация хранилась в виде отдельных компонентов, но реальная эксплуатация показала, что такая тонкая детализация, которая была первоначально задумана, отнюдь не требуется. Например, предположим, что таблица `Employee` включает столбцы, предназначенные для идентификации сотрудника, а также другие данные; с другой стороны, предусмотрена таблица `EmployeeIdentification`, которая специально предназначена для хранения только идентификационной информации.

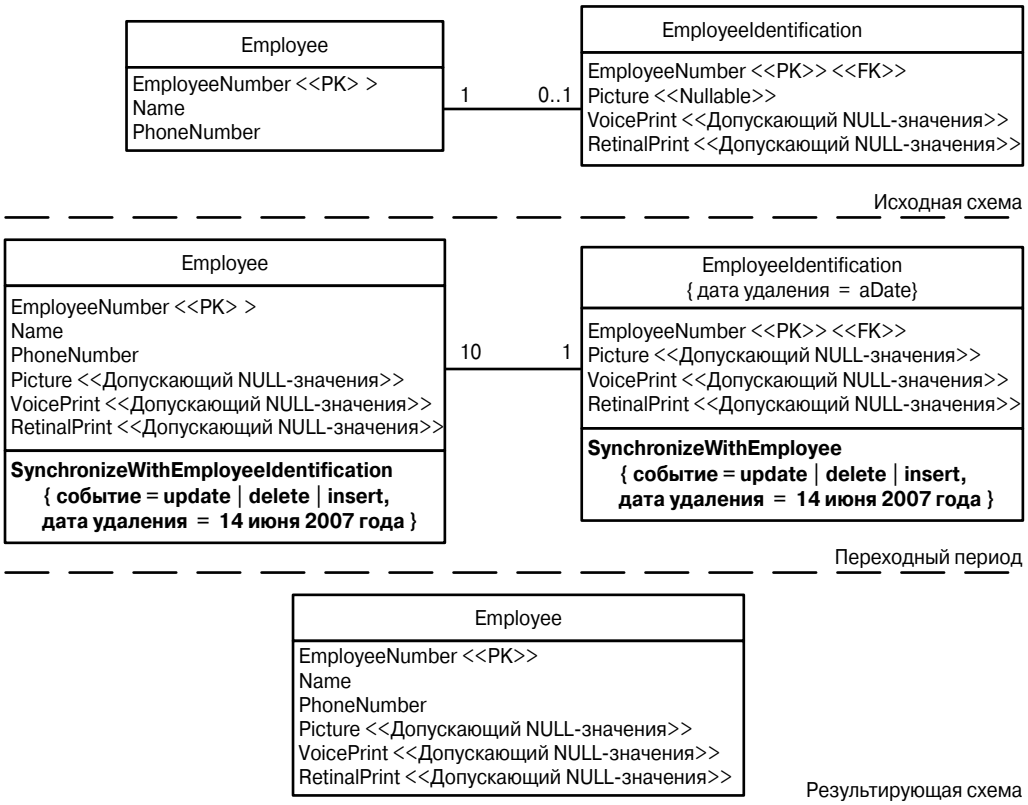


Рис. 6.7. Перемещение всех столбцов из таблицы `EmployeeIdentification` в таблицу `Employee`

- **В процессе эксплуатации области применения разных таблиц фактически стали одинаковыми.** Со времен способ использования двух или большего количества таблиц может измениться до такой степени, что несколько таблиц начнут применяться для одного и того же назначения. Может также оказаться, что между двумя таблицами имеется взаимно-однозначная связь; в таком случае иногда бывает целесообразно выполнить слияние таблиц, чтобы исключить необходимость применять к ним операцию соединения. Наглядным примером такой ситуации могут служить таблицы `Employee` и `EmployeeIdentification`, о которых речь шла перед этим. Первоначально для регистрации информации о сотрудниках использовалась таблица `Employee`, а затем была введена таблица `EmployeeIdentification`, предназначенная для хранения только идентификационных данных. Но некоторые сотрудники не принимали во внимание, что существует эта таблица, поэтому провели доработку таблицы `Employee` в целях хранения в ней указанных данных.
- **По ошибке был создан дубликат таблицы.** Могло оказаться так, что два или несколько разработчиков ввели в действие одинаковые таблицы, не зная о том, что это сделано также другими; такая ситуация встречается часто, если разработчики относятся к

разным коллективам или отсутствуют метаданные, описывающие схему таблицы. Например, предположим, что и таблица `FeeStructure`, и таблица `FeeSchedule` содержат данные об авансовом платеже, взимаемом банком при открытии текущего счета. Вторая таблица была добавлена, поскольку никто из разработчиков не смог со всей уверенностью ответить, для чего действительно предназначена таблица `FeeStructure`.

Потенциальные преимущества и недостатки

Объединение двух или нескольких таблиц может привести к потере точности данных после слияния таблиц, имеющих большую степень детализации. Кроме того, если осуществляется слияние таблиц, которые (по вашему мнению) используются для одной и той же цели, то вы подвергаетесь риску столкнуться с тем, что эти таблицы фактически используются для различных назначений. Например, предположим, что таблица `EmployeeIdentification` была введена в целях выделения информации, важной с точки зрения безопасности, в отдельную таблицу, к которой предоставляются ограниченные права доступа. В таком случае вы обнаружите, что должны повторно ввести одну или несколько исходных таблиц. Таким образом, решение о слиянии таблиц следует принимать с учетом области применения данных.

Процедура обновления схемы

Как показано на рис. 6.7, для обновления схемы базы данных при осуществлении операции “Слияние таблиц” необходимо выполнить два описанных ниже действия. Во-первых, требуется ввести в действие таблицу, формируемую путем слияния; в рассматриваемом примере необходимо ввести в таблицу `Employee` такие же столбцы, как в таблице `EmployeeIdentification`, применив команду `ADD COLUMN` языка SQL. При этом следует учитывать, что таблица `Employee` уже может включать некоторые или все требуемые столбцы, но в таком случае обычно обнаруживаются несовместимости или наличие чрезмерно загроможденного кода доступа к домену; указанная операция рефакторинга должна позволить упростить прикладной код. Во-вторых, необходимо ввести триггер (триггеры) синхронизации, который позволял бы обеспечить поддержку синхронизации таблиц друг с другом. Триггер (триггеры) должен вызываться при внесении любого изменения в столбцы и должен быть реализован таким образом, чтобы не возникали циклы. Иными словами, если произойдет изменение значения в одном из исходных столбцов, то должна быть обновлена таблица `Employee`, но это обновление не должно активизировать аналогичное обновление в исходных таблицах и т.д.

В примере, приведенном на рис. 6.7, показано, что в таблице `Employee` первоначально хранились данные о сотрудниках. Но со временем была также введена таблица `EmployeeIdentification`, предназначенная для хранения идентификационной информации, относящейся к сотрудникам. Поэтому было решено выполнить слияние таблиц `Employee` и `EmployeeIdentification`, чтобы иметь возможность хранить всю информацию, касающуюся сотрудников, в одном месте. Для этого был введен в действие триггер `SynchronizeIdentification`, который позволяет поддерживать синхронизацию значений в рассматриваемых таблицах. Ниже приведен код SQL, содержащий операторы DDL, которые позволяют ввести столбцы `Picture`, `VoicePrint`, `RetinalPrint`, а затем в конечном итоге удалить таблицу `EmployeeIdentification`.

```
ALTER TABLE Employee ADD Picture BINARY;

COMMENT ON Employee.Picture 'Added as the result of merging Employee
and EmployeeIdentification finaldate = December 14 2007';

ALTER TABLE Employee ADD VoicePrint BINARY;

COMMENT ON Employee.VoicePrint 'Added as the result of merging Employee
and EmployeeIdentification finaldate = December 14 2007';

ALTER TABLE Employee ADD RetinalPrint BINARY;

COMMENT ON Employee.RetinalPrint 'Added as the result of merging Employee
and EmployeeIdentification finaldate = December 14 2007';
```

Процедура переноса данных

Все данные из исходной таблицы (таблиц) необходимо скопировать в таблицу, создаваемую в результате слияния, в данном случае из таблицы `EmployeeIdentification` в таблицу `Employee`. Для выполнения этого действия может применяться несколько разных способов, например, основанных на использовании сценария SQL или какого-либо инструментального средства ETL (Extract–Transform–Load — извлечь–преобразовать–загрузить). (В рассматриваемом примере операции рефакторинга не должен предусматриваться шаг преобразования.)

Ниже приведен код SQL, содержащий операторы DDL, позволяющие осуществить объединение данных из таблиц `EmployeeIdentification` и `Employee`.

```
/* Одноразовый перенос данных из таблицы Employee в таблицу
EmployeeIdentification. Если обе таблицы активны, необходимо
предусмотреть триггер, который обеспечивает синхронизацию обеих таблиц
*/

          UPDATE Employee e SET e.Picture =
            (SELECT ei.Picture FROM EmployeeIdentificaion ei
              WHERE
                ei.EmployeeNumber = e.EmployeeNumber);

          UPDATE Employee e SET e.VoicePrint =
            (SELECT ei.VoicePrint FROM EmployeeIdentificaion ei
              WHERE
                ei.EmployeeNumber = e.EmployeeNumber);

          UPDATE Employee e SET e.RetinalPrint =
            (SELECT ei.RetinalPrint FROM EmployeeIdentificaion ei
              WHERE
      ei.EmployeeNumber = e.EmployeeNumber);

-- 14 декабря 2007 года
          DROP TRIGGER SynchronizeWithEmployee;
          DROP TRIGGER SynchronizeWithEmployeeIdentification;
          DROP TABLE EmployeeIdentification;
```

Ниже приведен код, который показывает, как используются триггеры SynchronizeWithEmployeeIdentification и SynchronizeWithEmployee для обеспечения синхронизации значений в рассматриваемых таблицах.

```
CREATE TRIGGER SynchronizeWithEmployeeIdentification
          BEFORE INSERT OR UPDATE OR DELETE
          ON Employee
          REFERENCING OLD AS OLD NEW AS NEW
          FOR EACH ROW
          DECLARE
          BEGIN
          IF updating THEN
            updateOrCreateEmployeeIdentification;
          END IF;
          IF inserting THEN
            createNewEmployeeIdentification;
          END IF;
          IF deleting THEN
            deleteEmployeeIdentification;
          END IF;
          END;
          /

          CREATE TRIGGER SynchronizeWithEmployee
          BEFORE INSERT OR UPDATE OR DELETE
          ON EmployeeIdentification
          REFERENCING OLD AS OLD NEW AS NEW
          FOR EACH ROW
          DECLARE
          BEGIN
          IF updating THEN
            updateOrCreateEmployee;
          END IF;
          IF inserting THEN
            createNewEmployee;
          END IF;
          IF deleting THEN
            deleteEmployee;
          END IF;
          END;
/
```

Процедура обновления программ доступа

Кроме этой очевидной доработки, необходимо перейти к применению таблицы Employee, а не прежней таблицы (таблиц); к числу потенциальных обновлений в основном относятся описанные ниже.

1. **Упрощение кода доступа к данным.** Часть существующего кода доступа может применяться исключительно в связи с осуществлением доступа к двум или нескольким таблицам, которые участвуют в слиянии. Например, может оказаться так, что класс Employee обновляет относящуюся к нему информацию в двух

таблицах, в которых эта информация хранится в настоящее время; но теперь произошло слияние этих двух таблиц в одну.

2. **Устранение неполных или противоречивых обновлений.** После того как все данные будут храниться в одном месте, может обнаружиться, что отдельные программы доступа к данным применяются для работы только с подмножествами данных. Например, предположим, что класс `Customer` в настоящее время обеспечивает обновление относящейся к нему информации о номерах домашних телефонов в двух таблицах, несмотря на то, что эта информация в действительности хранится в трех таблицах (которые теперь объединены в результате слияния в одну). А разработчики других программ могли прийти к выводу, что качество данных в третьей таблице недостаточно высоко, и в связи с этим включить код, позволяющий устранить эти проблемы. Например, в классе, применяемом для составления отчетов, может быть предусмотрено преобразование номеров телефонов, представленных как NULL-значения, в строковые значения "`Unknown`", а теперь, когда исключена возможность появления NULL-значений номеров телефонов, этот код может быть удален.
3. **Некоторые данные, полученные в результате слияния, не требуются для определенных программ доступа.** Предположим, что для некоторых программ доступа, которые работают в настоящее время с таблицей `Employee`, требуются только те данные, которые содержатся в исходном варианте этой таблицы. А после добавления столбцов из таблицы `EmployeeIdentification` возникают предпосылки того, что существующие программы доступа не будут обновлять эти новые столбцы должным образом. Поэтому, возможно, потребуется доработка существующих программ доступа, чтобы в них учитывалось наличие новых столбцов и обеспечивалась работа с ними. Например, предположим, что в исходную таблицу, применявшуюся в классе `Employee`, в результате слияния был включен столбец `BirthDate`. Поэтому класс `Employee` должен, как минимум, предотвращать перезапись значений этого столбца и замену недействительными данными, а также должен вставлять соответствующие значения при создании нового объекта клиента. К строкам, которые были введены в результате слияния в таблицу `Employee`, может также потребоваться применить операцию "Введение заданного по умолчанию значения" (с. 213).

Ниже приведен пример, который показывает, какие изменения кода могут быть введены в код при выполнении операции "Слияние таблиц" применительно к таблицам `Employee` и `EmployeeIdentification`.

```
// Код до рефакторинга
public Employee getEmployeeInformation (Long employeeNumber) throws
SQLException {
  Employee employee = new Employee();

  stmt.prepare(
  "SELECT EmployeeNumber, Name, PhoneNumber " +
  "FROM Employee" +
  "WHERE EmployeeNumber = ?");
  stmt.setLong(1,employeeNumber);
  stmt.execute();
```

```
  ResultSet rs = stmt.executeQuery();
  employee.setEmployeeNumber(rs.getLong("EmployeeNumber"));
  employee.setName(rs.getLong("Name"));
  employee.setPhoneNumber(rs.getLong("PhoneNumber"));

  stmt.prepare(
  "SELECT Picture, VoicePrint, RetinalPrint " +
  "FROM EmployeeIdentification" +
  "WHERE EmployeeNumber = ?");
  stmt.setLong(1,employeeNumber);
  stmt.execute();
  rs = stmt.executeQuery();
  employee.setPicture(rs.getBlob("Picture"));
  employee.setVoicePrint(rs.getBlob("VoicePrint"));
  employee.setRetinalPrint(rs.getBlob("RetinalPrint"));

  return employee;
}

// Код после рефакторинга
public Employee getEmployeeInformation (Long employeeNumber) throws
SQLException {
  Employee employee = new Employee();

  stmt.prepare(
  "SELECT EmployeeNumber, Name, PhoneNumber " +
  "Picture, VoicePrint, RetinalPrint "+
  "FROM Employee" +
  "WHERE EmployeeNumber = ?");
  stmt.setLong(1,employeeNumber);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  employee.setEmployeeNumber(rs.getLong("EmployeeNumber"));
  employee.setName(rs.getLong("Name"));
  employee.setPhoneNumber(rs.getLong("PhoneNumber"));
  employee.setPicture(rs.getBlob("Picture"));
  employee.setVoicePrint(rs.getBlob("VoicePrint"));
  employee.setRetinalPrint(rs.getBlob("RetinalPrint"));
  return employee;
}
```

Операция рефакторинга “Перемещение столбца”

Эта операция обеспечивает перемещение столбца таблицы вместе со всеми данными в другую существующую таблицу (рис. 6.8).

Обоснование

Необходимость в применении операции “Перемещение столбца” может быть обусловлена несколькими причинами. Первые две причины могут показаться противоречащими друг другу, но следует помнить, что необходимость в осуществлении рефакторинга

базы данных зависит от конкретной ситуации. Ниже перечислены основания, которые чаще всего влекут за собой необходимость применения операции "Перемещение столбца".

- **Нормализация.** Часто встречается такая ситуация, что в связи с наличием одного из столбцов нарушаются правила нормализации. Иногда перемещение такого столбца в другую таблицу позволяет повысить степень нормализации исходной таблицы и тем самым уменьшить избыточность данных в базе данных.
- **Денормализация в целях сокращения количества часто применяемых операций соединения.** Очень часто встречается такая ситуация, в которой обнаруживается, что какая-то таблица включается в соединение исключительно ради получения доступа к какому-то отдельному столбцу. В этом случае можно повысить производительность, исключив необходимость в выполнении операции соединения путем перемещения столбца в другую таблицу.
- **Реорганизация разделенной таблицы.** Иногда возникает такая ситуация, что ранее была выполнена операция "Разбиение таблицы" (с. 177) или таблица была фактически разделена в исходном проекте, а затем обнаружено, что один или несколько столбцов требует перемещения. Возможно также, что такой столбец присутствует в таблице, к которой часто осуществляется доступ, но сам столбец почти не используется, или, возможно, какой-то столбец находится в таблице, данные которой требуются редко, но необходимость в получении доступа к самому столбцу возникает весьма часто. В первом случае перемещение столбца способствует повышению производительности сети, поскольку исключается необходимость выполнять выборку данных столбца, а затем передавать эти данные в приложения, тогда как фактически они не требуются, а во втором случае повышается производительность базы данных, поскольку уменьшается потребность в выполнении операций соединения.

Потенциальные преимущества и недостатки

Перемещение столбца в целях повышения степени нормализации приводит к уменьшению избыточности данных, но может повлечь за собой снижение производительности, если в приложениях потребуются дополнительные соединения для получения данных. И наоборот, если повышение производительности будет достигнуто в результате денормализации схемы, возникающей после перемещения столбца, то увеличивается избыточность данных.

Процедура обновления схемы

Для обновления схемы базы данных при выполнении операции "Перемещение столбца" необходимо выполнить описанные ниже действия.

1. **Сформулировать правило (правила) удаления.** Для этого необходимо выяснить, что должно происходить при удалении любой строки из отдельной таблицы. В частности, необходимо определить, следует ли удалять соответствующую строку в другой таблице, вводить NULL-значение или нулевое значение в другую строку, задавать в качестве соответствующего значения то или иное значение, предусмотренное по умолчанию, или оставлять соответствующее значение неизменным. Это правило должно быть реализовано в коде триггера (как будет

описано ниже в этом разделе). Следует учитывать, что уже могут существовать от нуля или больше триггеров удаления, предназначенных для поддержки правил ссылочной целостности, распространяющихся на рассматриваемые таблицы.

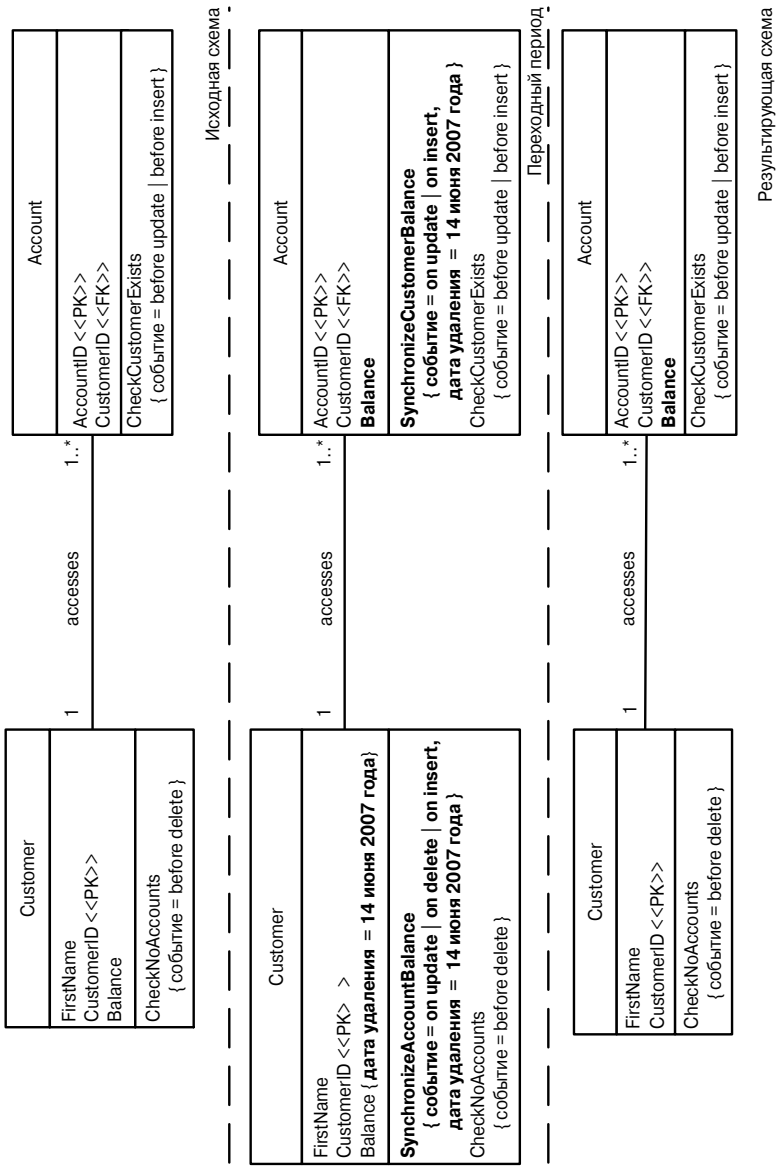


Рис. 6.8. Перемещение столбца `Balance` *из таблицы* `Customer` *в таблицу* `Account`

1. **Сформулировать правило (правила) вставки.** Для этого необходимо выяснить, что должно происходить при вставке строки в одну из таблиц. В частности, необходимо определить, должна ли выполняться вставка соответствующей строки в другую таблицу или не должно происходить ничего. Это правило должно быть реализовано в коде триггера, но могут уже существовать от нуля или больше триггеров вставки.
2. **Ввести новый столбец.** Добавьте столбец к целевой таблице с помощью команды `ADD COLUMN` языка SQL. На рис. 6.8 таковым является столбец `Account.Balance`.
3. **Ввести триггеры.** Триггеры, предназначенные для копирования данных из одного столбца в другой в течение переходного периода, необходимо предусмотреть и на исходном, и на новом столбце. Такой триггер (триггеры) должен вызываться при любом изменении в строке.

На рис. 6.8 показан пример, в котором столбец `Customer.Balance` перемещается в таблицу `Account`. Эта операция перемещения применяется для решения проблемы нормализации, так как вместо сохранения баланса после каждого обновления данных, относящихся к любому счету клиента, можно сохранить эти данные по одному разу для каждого отдельного счета. В течение переходного периода, как и следует ожидать, столбец `Balance` присутствует и в таблице `Customer`, и в таблице `Account`.

Интерес представляют и существующие триггеры. В таблице `Account` уже имелся триггер для вставок и обновлений, который обеспечивал проверку того, существует ли соответствующая строка в таблице `Customer`, т.е. выполнял элементарную проверку ссылочной целостности (Referential Integrity — RI). Этот триггер остается неизменным. А в таблице `Customer` имелся триггер удаления, который гарантировал, чтобы в этой таблице не производилось удаление строки, если на нее ссылается строка таблицы `Account`; это — еще одна проверка ссылочной целостности. Преимущество данного подхода состоит в том, что не требуется реализовывать правило удаления для перемещаемого столбца, поскольку невозможно "допустить ошибку" и удалить строку из таблицы `Customer`, если на нее ссылается одна или несколько строк таблицы `Account`.

Ниже приведен код, в котором вводится столбец `Account.Balance` и задаются триггеры `SynchronizeCustomerBalance` и `SynchronizeAccountBalance`, позволяющие синхронизировать данные в столбцах `Balance`. В состав этого кода входят также сценарии, позволяющие удалить приведенный здесь вспомогательный код по окончании переходного периода.

```
ALTER TABLE Account ADD Balance NUMBER(32,7);
COMMENT ON Account.Balance 'Moved from Customer table, finaldate =
14 июня 2007 года';

COMMENT ON Customer.Balance 'Moved to Account table, dropdate =
14 июня 2007 года';

CREATE OR REPLACE TRIGGER SynchronizeCustomerBalance
  BEFORE INSERT OR UPDATE
  ON Account
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
```

```
  DECLARE
  BEGIN
    IF :NEW.Balance IS NOT NULL THEN
      UpdateCustomerBalance;
    END IF;
  END;

CREATE OR REPLACE TRIGGER SynchronizeAccountBalance
  BEFORE INSERT OR UPDATE OR DELETE
  ON Customer
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF DELETING THEN
      DeleteCustomerIfAccountNotFound;
    END IF;
    IF (UPDATING OR INSERTING) THEN
      IF :NEW.Balance IS NOT NULL THEN
        UpdateAccountBalanceForCustomer;
      END IF;
    END IF;
  END;

-- 14 июня 2007 года
ALTER TABLE Customer DROP COLUMN Balance;
DROP TRIGGER SynchronizeCustomerBalance;
DROP TRIGGER SynchronizeAccountBalance;
```

Процедура переноса данных

Скопируйте все данные из исходного столбца в новый столбец, в данном случае из столбца `Customer.Balance` в столбец `Account.Balance`. Для выполнения этого действия может применяться несколько разных способов, например, основанных на использовании сценария SQL или какого-либо инструментального средства ETL. (В рассматриваемом примере операции рефакторинга не должен предусматриваться шаг преобразования.) В следующем примере кода приведены операторы DML, позволяющие переместить значения столбца `Balance` из таблицы `Customer` в таблицу `Account`:

```
/* Одноразовый перенос данных из таблицы Customer.Balance в таблицу
Account.Balance. Если оба столбца активизированы, необходимо
предусмотреть триггер, который обеспечивает синхронизацию обоих
столбцов баланса */

UPDATE Account SET Balance =
  (SELECT Balance FROM Customer
    WHERE CustomerID = Account.CustomerID);
```

Процедура обновления программ доступа

Необходимо тщательно проанализировать программы доступа, а затем обновить их должным образом в течение переходного периода. К числу потенциальных обновлений в основном относятся описанные ниже.

1. **Переопределение операторов соединений в целях использования перемещаемого столбца.** Рефакторингу должны быть подвергнуты все операторы соединения, реализованные с помощью программных конструкций SQL или метаданных, для обеспечения работы с перемещаемым столбцом. Например, после перемещения данных из столбца `Customer.Balance` в столбец `Account.Balance` может потребоваться откорректировать применяемые запросы для получения информации о балансе из таблицы `Account`, а не `Customer`.
2. **Учесть в операторах соединения наличие новой таблицы.** Теперь в операторы соединения должна быть включена таблица `Account`, если такая доработка еще не выполнена. Но это может привести к снижению производительности.
3. **Удалить исходную таблицу из операторов соединения.** В некоторых операторах соединения таблица `Customer` может быть введена с единственной целью — включить в соединение данные из столбца `Customer.Balance`. А теперь, после перемещения указанного столбца, может появиться возможность исключить упоминания о таблице `Customer` из операторов соединения, что может в принципе способствовать повышению производительности.

Ниже приведен код, которые показывает, что в исходном варианте кода применялась ссылка на столбец `Customer.Balance`, а в обновленном коде для работы используется столбец `Account.Balance`.

```
// Код до рефакторинга
      public BigDecimal getCustomerBalance(Long customerId) throws
SQLException {
        PreparedStatement stmt = null;
        BigDecimal customerBalance = null;
        stmt = DB.prepare("SELECT Balance FROM Customer " +
                "WHERE CustomerId = ?");
        stmt.setLong(1, customerId.longValue());
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
          customerBalance = rs.getBigDecimal("Balance");
        }
        return customerBalance;
}

// Код после рефакторинга
      public BigDecimal getCustomerBalance(Long customerId) throws
SQLException {
        PreparedStatement stmt = null;
        BigDecimal customerBalance = null;
        stmt = DB.prepare(
           "SELECT SUM(Account.Balance) Balance " +
           "FROM Customer, Account " +
           "WHERE Customer.CustomerId= Account.CustomerId " +
```

```
            "AND CustomerId = ?");
        stmt.setLong(1, customerId.longValue());
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
          customerBalance = rs.getBigDecimal("Balance");
        }
        return customerBalance;
}
```

Операция рефакторинга “Переименование столбца”

Эта операция используется для переименования существующего столбца таблицы (рис. 6.9).

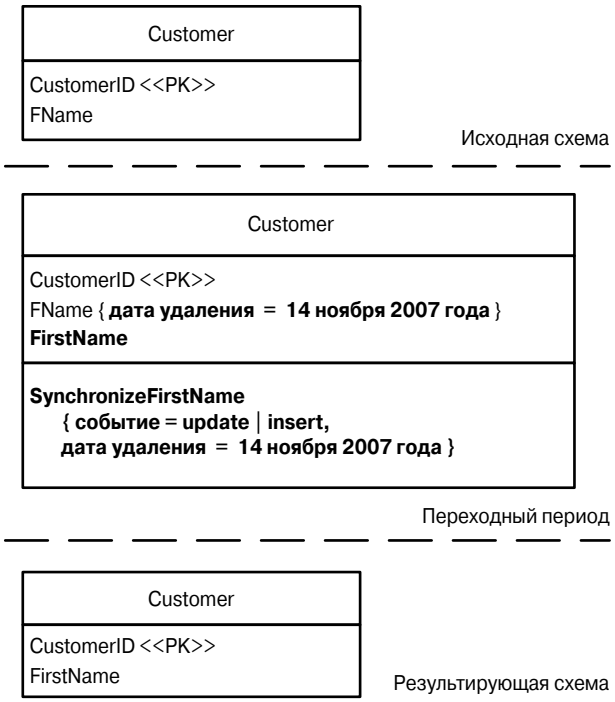


Рис. 6.9. Переименование столбца `Customer.FName`

Обоснование

Основные причины применения операции “Переименование столбца” состоят в том, что может потребоваться повышение удобства для чтения используемой схемы базы данных, обеспечение соответствия с принятыми на предприятии соглашениями об именовании объектов базы данных или создание предпосылок для переноса базы данных на другую платформу. Например, если базу данных необходимо перенести с одного программного продукта баз данных на другой, может оказаться, что одно из первоначальных имен столбцов нельзя использовать, поскольку оно совпадает с зарезервированным ключевым словом в новом программном обеспечении баз данных.

Потенциальные преимущества и недостатки

При оценке основных потенциальных преимуществ и недостатков необходимо учитывать, оправдывает ли достигнутое повышение удобства для чтения и (или) согласованности благодаря введению нового имени те затраты, на которые придется пойти для осуществления рефакторинга внешних приложений.

Процедура обновления схемы

Для переименования столбца необходимо выполнить несколько действий.

1. **Ввести новый столбец.** На рис. 6.9 показано, что прежде всего с помощью команды `ADD COLUMN` языка SQL в целевую таблицу добавляется столбец `FirstName`.
2. **Ввести триггер синхронизации.** Как показано на рис. 6.9, для копирования данных из одного столбца в другой в течение переходного периода требуется триггер, который должен вызываться при любом изменении в строке данных.
3. **Переименовать другие столбцы.** Например, если в других таблицах в качестве внешнего ключа (или его части) используется столбец `FName`, то может потребоваться рекурсивно применить операцию "Переименование столбца" для обеспечения согласованности имен. В частности, если столбец `Customer.CustomerNumber` должен быть переименован в `Customer.CustomerID`, то в связи с этим может возникнуть необходимость переименовать все столбцы `CustomerNumber` в других таблицах. Таким образом, в связи с этим нововведением столбец `Account.CustomerNumber` теперь будет переименован в `Account.CustomerID` для обеспечения согласованности имен столбцов.

В приведенном ниже коде содержатся операторы DDL, которые позволяют переименовать столбец `Customer.FName` в `Customer.FirstName`, создать триггер `SynchronizeFirstName`, предназначенный для синхронизации данных в течение переходного периода, и удалить исходный столбец и триггер по истечении переходного периода.

```
ALTER TABLE Customer ADD FirstName VARCHAR(40);

COMMENT ON Customer.FirstName 'Renaming of FName column, finaldate =
November 14 2007';

COMMENT ON Customer.FName 'Renamed to FirstName, dropdate = November
14 2007';

UPDATE Customer SET FirstName = FName;

CREATE OR REPLACE TRIGGER SynchronizeFirstName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF INSERTING THEN
    IF :NEW.FirstName IS NULL THEN
      :NEW.FirstName := :NEW.FName;
```

```
    END IF;
    IF :NEW.Fname IS NULL THEN
      :NEW.FName := :NEW.FirstName;
    END IF;
  END IF;

  IF UPDATING THEN
    IF NOT(:NEW.FirstName=:OLD.FirstName) THEN
      :NEW.FName:=:NEW.FirstName;
    END IF;
    IF NOT(:NEW.FName=:OLD.FName) THEN
      :NEW.FirstName:=:NEW.FName;
    END IF;
  END IF;
  END;
/

-- 30 ноября 2007 года
DROP TRIGGER SynchronizeFirstName;
ALTER TABLE Customer DROP COLUMN FName;
```

Процедура переноса данных

Необходимо скопировать все данные из исходного столбца в новый столбец, в рассматриваемом случае из столбца `FName` в столбец `FirstName`. Дополнительные сведения по этой теме приведены в описании операции рефакторинга “Перемещение данных” (с. 219).

Процедура обновления программ доступа

Необходимо обновить внешние программы, которые ссылаются на столбец `Customer.FName`, чтобы этот столбец упоминался в них под новым именем. Для этого достаточно обновить все внедренные операторы SQL и (или) метаданные отображения. Ниже приведены файлы Hibernate-отображения, которые показывают, как изменятся файлы отображения при переименовании столбца `fName`.

```
// Отображение до рефакторинга
<hibernate-mapping>
<class name="Customer" table="Customer">
  <id name="id" column="CUSTOMERID">
      <generator class="CustomerIdGenerator"/>
  </id>
  <property name="fName"/>
</class>
</hibernate-mapping>

// Промежуточное отображение
<hibernate-mapping>
<class name="Customer" table="Customer">
  <id name="id" column="CUSTOMERID">
      <generator class="CustomerIdGenerator"/>
  </id>
  <property name="fName"/>
```

```
  <property name="firstName"/>
</class>
</hibernate-mapping>

// Отображение после рефакторинга
<hibernate-mapping>

<class name="Customer" table="Customer">
  <id name="id" column="CUSTOMERID">
      <generator class="CustomerIdGenerator"/>
  </id>
  <property name="firstName"/>
</class>
</hibernate-mapping>
```

Операция рефакторинга “Переименование таблицы”

Эта операция применяется для переименования существующей таблицы (рис. 6.10 и 6.11).

Обоснование

Операция “Переименование таблицы” главным образом применяется для того, чтобы пояснить назначение таблицы и общую цель ее использования во всей схеме базы данных или привести имя таблицы в соответствие с принятыми соглашениями об именовании объектов базы данных. В идеальном случае обе эти причины должны иметь одинаковое обоснование.

Потенциальные преимущества и недостатки

При оценке основных потенциальных преимуществ и недостатков необходимо учитывать, оправдывает ли достигнутое повышение удобства для чтения и (или) согласованности благодаря введению нового имени те затраты, на которые придется пойти для осуществления рефакторинга внешних приложений, которые обращаются к этой таблице.

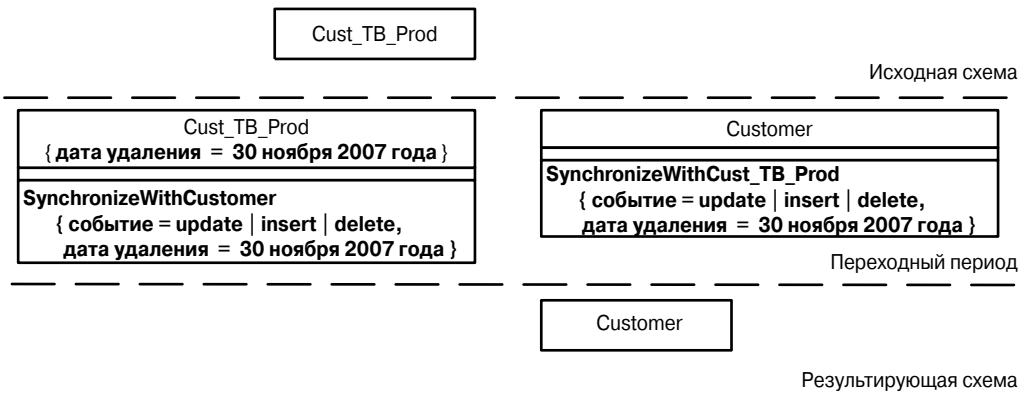


Рис. 6.10. Переименование таблицы `Cust_TB_Prod` *в* `Customer`

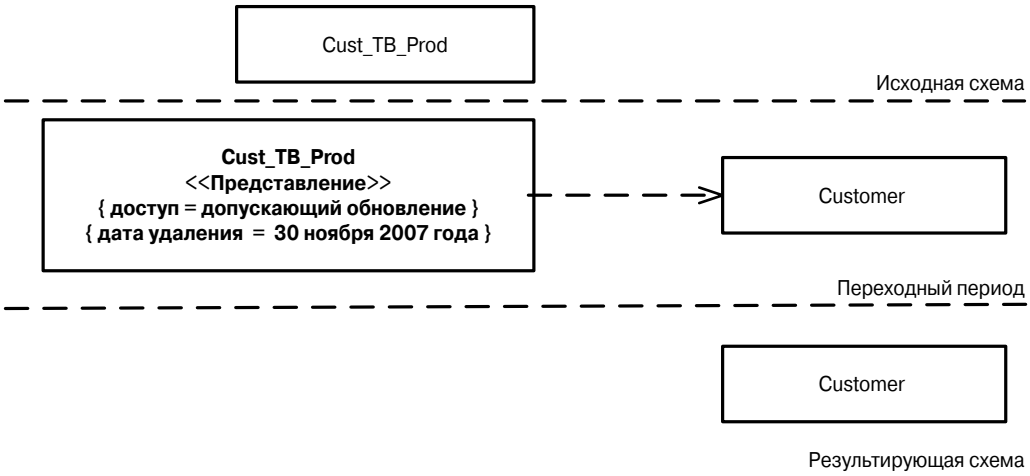


Рис. 6.11. Переименование таблицы `Cust_TB_Prod` в `Customer` с помощью представления

Процедура обновления схемы путем введения новой таблицы

Для осуществления операции “Переименование таблицы” создается новая таблица с использованием команды `CREATE TABLE` языка SQL, в рассматриваемом случае — таблица `Customer`. Если какие-либо столбцы таблицы `Cust_TB_Prod` применяются в других таблицах в составе внешнего ключа (или его части), то необходимо подвергнуть рефакторингу эти ограничения и (или) индексы, реализующие внешний ключ, чтобы они ссылались на таблицу `Customer`.

Как показано на рис. 6.10, необходимо переименовать таблицу `Cust_TB_Prod` в `Customer`. Триггеры `SynchronizeCust_TB_Prod` и `SynchronizeCustomer` поддерживают синхронизацию двух таблиц друг с другом. Каждый триггер вызывается при любом изменении в строке таблицы `Cust_TB_Prod` или `Customer` соответственно. В приведенном ниже коде содержатся операторы DDL, предназначенные для переименования таблицы и введения триггеров.

```
CREATE TABLE Customer
  (FirstName VARCHAR(40),
  LastName VARCHAR(40),
);

COMMENT ON Customer 'Renaming of Cust_TB_Prod, finaldate = September
14 2006';

COMMENT ON Cust_TB_Prod 'Renamed to Customer, dropdate = September 14
2006';

CREATE OR REPLACE TRIGGER SynchronizeCustomer
BEFORE INSERT OR UPDATE
ON Cust_TB_Prod
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
```

```
BEGIN
IF updating THEN
  findAndUpdateIfNotFoundCreateCustomer;
END IF;
IF inserting THEN
  createNewIntoCustomer;
END IF;
IF deleting THEN
  deleteFromCustomer;
END IF;
END;
/
```

```
CREATE OR REPLACE TRIGGER SynchronizeCust_TB_Prod
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
IF updating THEN
        findAndUpdateIfNotFoundCreateCust_TB_Prod;
END IF;
IF inserting THEN
  createNewIntoCust_TB_Prod;
END IF;
IF deleting THEN
  deleteFromCust_TB_Prod;
END IF;
END;
/
```

Процедура обновления схемы с помощью обновляемого представления

Второй подход состоит в том, что должна быть переименована таблица, а затем введено обновляемое представление с именем исходной таблицы. Следует отметить, что некоторые программные продукты баз данных поддерживают опцию `RENAME` команды `ALTER TABLE`. Если в применяемом программном продукте это не предусмотрено, то необходимо повторно создать ту же таблицу с новым именем, а затем загрузить в нее данные. Следует также запланировать удаление старой таблицы, чтобы исключить возможность ее случайного использования кем-то. Как показано на рис. 6.11, обновляемое представление требуется для использования в течение переходного периода для обеспечения работы внешних программ доступа, которые еще не были подвергнуты рефакторингу в целях применения переименованной таблицы. Такая стратегия осуществима только в том случае, если база данных поддерживает обновляемые представления.

На рис. 6.11 показано, как переименовать таблицу с использованием представления. Как показано ниже, достаточно воспользоваться конструкцией `RENAME TO` команды `ALTER TABLE` языка SQL, чтобы переименовать таблицу и создать представление.

```
ALTER TABLE Cust_tb_Prod RENAME TO Customer;
```

```
CREATE VIEW Cust_tb_Prod AS
  SELECT * FROM Customer;
```

Как и при использовании подхода, основанного на создании новой таблицы, если какие-либо столбцы таблицы Cust_TB_Prod применяются в других таблицах в составе внешнего ключа (или его части), то необходимо обновить эти ограничения и (или) индексы, реализующие внешний ключ, чтобы они ссылались на таблицу Customer.

Процедура переноса данных

Если используется подход, основанный на обновляемом представлении, то необходимость в перемещении данных отсутствует. Но при использовании подхода, основанного на создании новой таблицы, следует вначале скопировать данные. Скопируйте все данные из исходной таблицы в новую таблицу, в этом случае из Cust_TB_Prod в Customer. Кроме того, необходимо ввести триггеры и на исходной, и на новой таблице, чтобы копировать данные из одной таблицы в другую в течение переходного периода. Эти триггеры должны вызываться при внесении любых изменений в таблицы. Триггеры должны быть реализованы таким образом, чтобы не возникали циклы; иными словами, если изменяется значение в таблице Cust_TB_Prod, то должна обновляться и таблица Employee, но указанное обновление не должно активизировать аналогичное обновление в таблице Cust_TB_Prod и т.д. В следующем коде показано, как скопировать данные из таблицы Cust_TB_Prod в таблицу Customer:

```
INSERT INTO Customer
  SELECT * FROM CUST_TB_PROD;
```

Процедура обновления программ доступа

Рефакторингу должны быть подвергнуты программы внешнего доступа для обеспечения работы с таблицей Customer, а не Cust_TB_Prod. В следующем Hibernate-отображении показаны изменения, которые должны быть внесены в связи с переименованием таблицы Cust_TB_Prod:

```
// Отображение до рефакторинга
<hibernate-mapping>
<class name="Customer" table="Cust_TB_Prod">
.....
</class>
</hibernate-mapping>

// Отображение после рефакторинга
<hibernate-mapping>
<class name="Customer" table="Customer">
.....
</class>
</hibernate-mapping>
```

Операция рефакторинга “Переименование представления”

Эта операция применяется для переименования существующего представления (рис. 6.12).

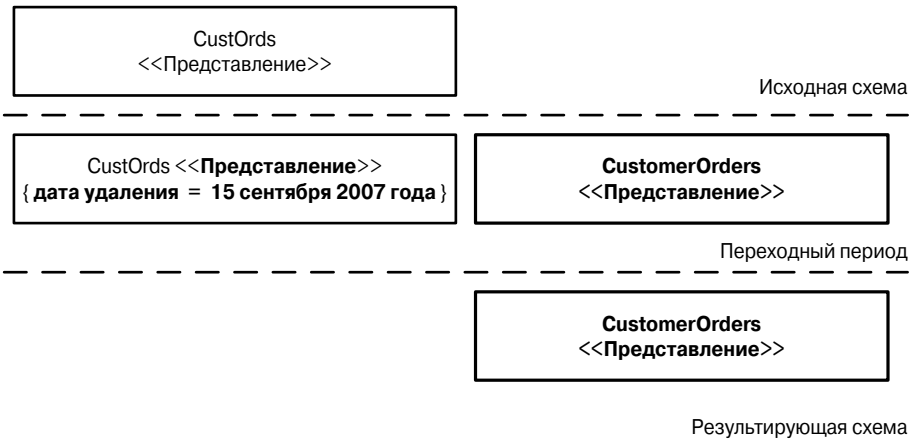


Рис. 6.12. Переименование представления `CustOrds` *в* `CustomerOrders`

Обоснование

Основные причины применения операции “Переименование представления” состоят в том, что может потребоваться повышение удобства для чтения используемой схемы базы данных или обеспечение соответствия с принятыми на предприятии соглашениями об именовании объектов базы данных. В идеальном случае обе эти причины должны иметь одинаковое обоснование.

Потенциальные преимущества и недостатки

При оценке основных потенциальных преимуществ и недостатков необходимо учитывать, оправдывает ли достигнутое повышение удобства для чтения и (или) согласованности благодаря введению нового имени те затраты, на которые придется пойти для осуществления рефакторинга внешних приложений, которые обращаются к этой таблице.

Процедура обновления схемы

Для выполнения операции “Переименование представления” необходимо осуществить описанные ниже действия.

1. **Ввести новое представление.** Создайте новое представление с использованием команды `CREATE VIEW` языка SQL. На рис. 6.12 таковым является `CustomerOrders`, определение которого должно быть согласовано с представлением `CustOrds`.

2. **Обозначить исходное представление как устаревшее.** После создания представления `CustomerOrders` необходимо указать, что `CustOrds` уже не подлежит обновлению при введении в действие новых средств или исправлении ошибок.
3. **Переопределить старое представление.** Необходимо переопределить представление `CustOrds` как основанное на `CustomerOrders`, чтобы предотвратить дублирование потоков кода. Преимущество этого действия состоит в том, что любые изменения в представлении `CustomerOrders`, такие как переход к использованию нового источника данных для столбца, будут распространяться на `CustOrds` без какой-либо дополнительной работы.

В следующем коде приведены операторы DDL, предназначенные для создания представления `CustomerOrders`, идентичные коду, который использовался для создания представления `CustOrds`:

```
CREATE VIEW CustomerOrders AS
SELECT
  Customer.CustomerCode,
  Order.OrderID,
  Order.OrderDate,
  Order.ProductCode
FROM Customer,Order
WHERE
  Customer.CustomerCode = Order.CustomerCode
  AND Order.ShipDate = TOMORROW
;

COMMENT ON CustomerOrders 'Renamed from CustOrds, CustOrds dropdate =
September 15 2007';
```

А следующий код показывает, как удалить, а затем повторно создать представление `CustOrds` для того, чтобы оно получало свои результаты из представления `CustomerOrders`:

```
DROP VIEW CustOrds;

CREATE VIEW CustOrds AS
SELECT CustomerCode, OrderID, OrderDate, ProductCode
FROM CustomerOrders
;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных не требуется переносить какие-либо данные.

Процедура обновления программ доступа

Необходимо подвергнуть рефакторингу все внешние программы, которые в настоящее время получают доступ к представлению `CustOrds`, чтобы они имели доступ к представлению `CustomerOrders`. В случае использования кода SQL, реализованного исключительно с помощью программных конструкций, такой рефакторинг сводится к обновлению конструкций `FROM/INTO`, а при использовании подходов, управляемых мета-

данными, уточняется имя в той части метаданных, которые относятся к этому представлению.

В следующем коде показано, как должна изменяться в определении представления `CustOrds` ссылка на представление `CustomerOrders`:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT * " +
  "FROM CustOrds "+
  "WHERE CustomerId = ?");
  stmt.setLong(1,customer.getCustomerID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
"SELECT * " +
"FROM CustomerOrders " +
"WHERE " +
"  CustomerId = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга "Замена данных типа LOB таблицей"

Эта операция предназначена для замены столбца с крупным объектом (Large OBject — LOB), который содержит структурированные данные, новой таблицей или столбцами, введенными в ту же таблицу (рис. 6.13). Данные типа LOB обычно хранятся в виде крупного двоичного объекта (Binary Large OBject — BLOB), строки символов переменной длины (`VARCHAR`), а в некоторых случаях — в виде данных XML.

Обоснование

Основная причина замены данных типа LOB таблицей может быть обусловлена необходимостью рассматривать части LOB как отдельные элементы данных. Такая ситуация очень часто встречается при использовании структур данных XML, которые в свое время были сохранены в виде одного столбца, в основном для предотвращения "разделения" структуры на отдельные столбцы.

Потенциальные преимущества и недостатки

Преимущество хранения сложной структуры данных в одном столбце состоит в том, что обеспечивается возможность легко извлечь конкретную структуру данных без особых затрат времени. Это преимущество оказывается особенно ценным, если с рассматриваемой структурой данных уже работает существующий код, а необходимость в использовании базы данных связана с потребностью иметь удобный механизм хранения файлов. Но если структура данных, содержащихся в столбце, имеющем тип данных LOB, очень сложна, то замена LOB таблицей или, возможно, несколькими таблицами позволит упростить работу с отдельными элементами данных, хранящимися в базе данных. Кроме того,

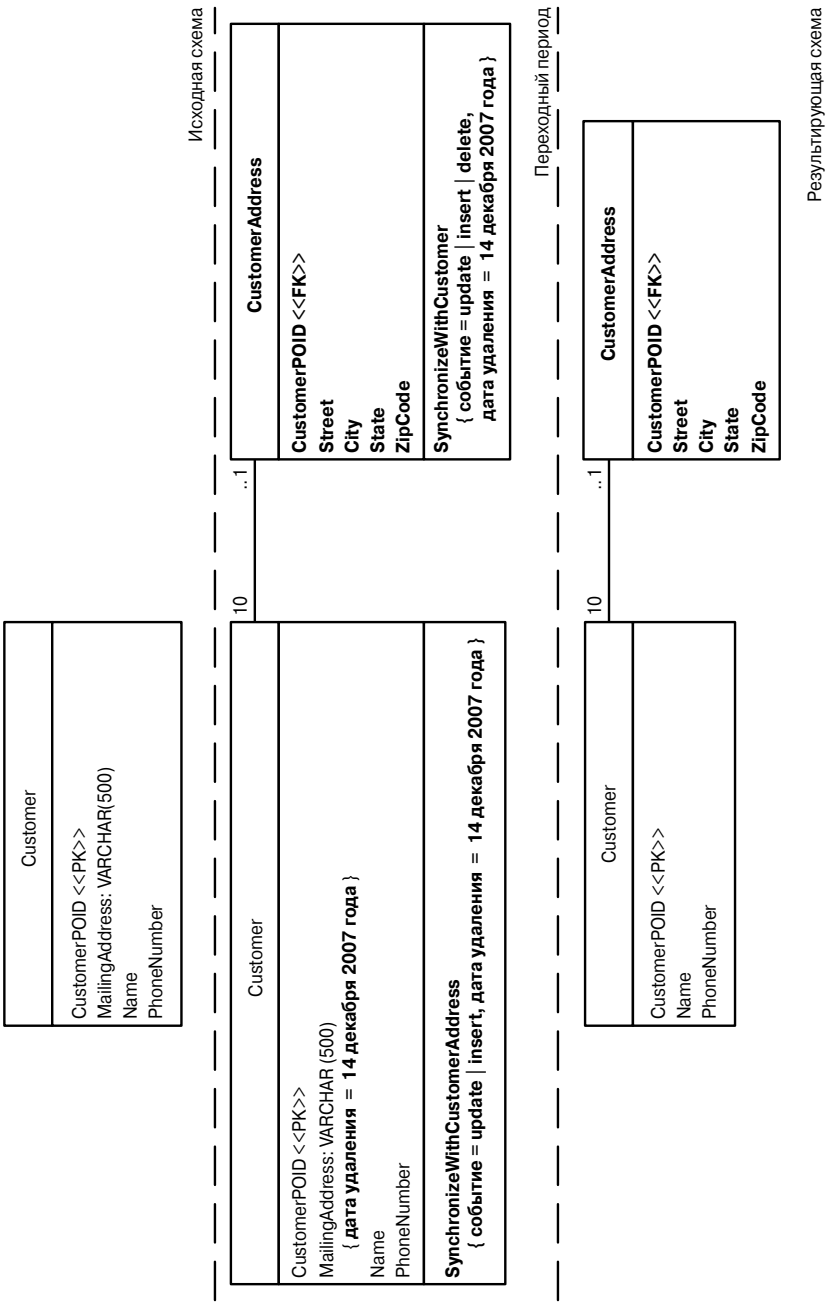


Рис. 6.13. Замена данных типа LOB таблицей

данные становятся более доступными для других приложений, в которых, возможно, не требуется учитывать все нюансы структуры данных, хранящихся в столбце типа LOB. Более того, если столбец типа LOB содержит некоторые данные, которые уже присутствуют в базе данных, то можно в принципе использовать эти существующие источники данных для представления соответствующих частей данных типа LOB, что способствует сокращению избыточности данных (и тем самым уменьшению количества ошибок, связанных с нарушением целостности данных). Недостаток этого подхода состоит в том, что он требует дополнительных затрат времени и выполнения сложных действий для разделения данных в целях дальнейшего их хранения в базе данных; кроме того, усложняются выборка данных, а также выполнение обратного преобразования в требуемую структуру.

Процедура обновления схемы

Как показано на рис. 6.13, применение операции "Замена данных типа LOB таблицей" является несложным. Для этого необходимо выполнить следующие действия.

1. **Определить схему таблицы.** В данном случае требуется проанализировать столбец `Customer.MailingAddress`, чтобы определить, какие данные в нем содержатся, а затем разработать схему таблицы, предназначенной для хранения этих данных. Если структура данных, содержащихся в столбце `MailingAddress`, является сложной, то необходимо либо предусмотреть в новой таблице меньшие столбцы типа LOB, либо рекурсивно применять операцию "Замена данных типа LOB таблицей" для преобразования более мелких структур, образовавшихся после разделения крупной структуры.
2. **Добавить таблицу.** На рис. 6.13 таковой является `CustomerAddress`. Столбцами этой таблицы становятся первичный ключ таблицы `Customer`, столбец `CustomerPOID` и новые столбцы, содержащие данные из столбца `MailingAddress`.
3. **Обозначить исходный столбец как устаревший.** Столбец `MailingAddress` должен быть отмечен как предназначенный для удаления в конце того периода, в течение которого сохраняются устаревшие столбцы.
4. **Добавить новый индекс.** В целях повышения производительности может потребоваться ввести новый индекс для таблицы `CustomerAddress` с помощью команды `CREATE INDEX`.
5. **Ввести триггеры синхронизации.** Для таблицы `Customer` требуется триггер, который позволил бы заполнять значения в таблице `CustomerAddress` должным образом. Этот триггер должен обеспечивать разделение структуры `MailingAddress` и правильное сохранение ее компонентов. Аналогичным образом, должен быть задан триггер на таблице `CustomerAddress`, предназначенный для обновления таблицы `Customer` в течение переходного периода.

Ниже приведен код, предназначенный для создания таблицы `CustomerAddress`, введения индекса, определения триггеров синхронизации и в конечном итоге удаления столбца и триггеров.

```
CREATE TABLE CustomerAddress (
  CustomerPOID NUMBER NOT NULL,
  Street VARCHAR(40),
```

```
  City VARCHAR(40),
  State VARCHAR(40),
  ZipCode VARCHAR(10)
);

CREATE INDEX IndexCustomerAddress ON
  CustomerAddress(CustomerPOID);

-- Триггеры, обеспечивающие синхронизацию таблиц
CREATE OR REPLACE TRIGGER SynchronizeWithCustomerAddress
BEFORE INSERT OR UPDATE OR DELETE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF updating THEN
    FindOrCreateCustomerAddress;
  END IF;
  IF inserting THEN
    CreateCustomerAddress;
  END IF;
  IF deleting THEN
    DeleteCustomerAddress;
  END IF;
END;
/

CREATE OR REPLACE TRIGGER SynchronizeWithCustomer
BEFORE INSERT OR UPDATE OR DELETE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF updating OR inserting THEN
    FindAndUpdateCustomer;
  END IF;
  IF deleting THEN
    UpdateCustomerNullAddress;
  END IF;
END;
/

-- 14 декабря 2007 года
DROP TRIGGER SynchronizeWithCustomerAddress;
DROP TRIGGER SynchronizeWithCustomer;

ALTER TABLE Customer DROP COLUMN MailingAddress;
```

Процедура переноса данных

Заполнение таблицы `CustomerAddress` должно осуществляться путем разделения, а затем копирования данных, содержащихся в столбце `Customer.MailingAddress`. Кроме того, для сопровождения связи необходимо также копировать значение `Customer.CustomerPOID`.

Если столбец `MailingAddress` содержит NULL-значение или не заполнен, то нет необходимости создавать строку в таблице `CustomerAddress`, соответствующую пустому полю.

Как показано в приведенном ниже коде, заполнение таблицы `CustomerAddress` можно обеспечить с помощью одного или нескольких сценариев SQL.

```
INSERT INTO CustomerAddress
  SELECT
    CustomerPOID,
    ExtractStreet(MailingAddress),
    ExtractCity(MailingAddress),
    ExtractState(MailingAddress),
    ExtractZipCode(MailingAddress)
  FROM Customer
    WHERE MailingAddress IS NOT NULL;
```

Процедура обновления программ доступа

Необходимо выявить наличие всех внешних программ, ссылающихся на столбец `Customer.MailingAddress`, чтобы можно было обновить их для обеспечения работы с таблицей `CustomerAddress` должным образом. В частности, необходимо выполнить описанные ниже действия.

1. **Удалить код преобразования.** Внешние программы могут содержать код, предназначенный для разделения данных, которые получены из столбца `MailingAddress`, для дальнейшей работы с субэлементами данных; внешние программы могут также содержать код, который принимает исходные элементы данных и создает строку в формате, предназначенном для хранения в столбце `MailingAddress`. После ввода в действие новой структуры данных этот код больше не потребуется.
2. **Ввести код преобразования.** И наоборот, для некоторых внешних программ может потребоваться точное воссоздание структуры данных, содержавшихся в столбце `MailingAddress`. Если такая потребность возникает применительно к нескольким приложениям, то целесообразно подумать о введении хранимой процедуры или предусмотреть применение какой-то библиотеки в сочетании с базой данных, позволяющей выполнять это преобразование и обеспечивающей повторное использование.
3. **Написать код, предназначенный для получения доступа к новой таблице.** После введения в действие таблицы `CustomerAddress` необходимо написать прикладной код, в котором используется эта новая таблица, а не таблица `MailingAddress`.

В следующем коде показано, как заменить программные конструкции, применявшиеся для выборки атрибутов данных из столбца `Customer.MailingAddress`, оператором `SELECT`, применяемым к таблице `CustomerAddress`:

```
// Код до рефакторинга
public Customer findByCustomerID(Long customerPOID) {
  Customer customer = new Customer();
  stmt = DB.prepare("SELECT CustomerPOID, "+
    "MailingAddress, Name, PhoneNumber " +
    "FROM Customer " +
    "WHERE CustomerPOID = ?");
  stmt.setLong(1, customerPOID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerId(rs.getLong("CustomerPOID"));
    customer.setName(rs.getString("Name"));
    customer.setPhoneNumber(rs.getString("PhoneNumber"));
    String mailingAddress = rs.getString("MailingAddress");
    customer.setStreet(extractStreet(mailingAddress));
    customer.setCity(extractCity(mailingAddress));
    customer.setState(extractState(mailingAddress));
    customer.setZipCode(extractZipCode(mailingAddress));
  }
return customer;
}

// Код после рефакторинга
public Customer findByCustomerID(Long customerPOID) {
  Customer customer = new Customer();
  stmt = DB.prepare("SELECT CustomerPOID, "+
    "Name, PhoneNumber, "+
    "Street, City, State, ZipCode " +
  "FROM Customer, CustomerAddress " +
  "WHERE Customer.CustomerPOID = ? " +
  "AND Customer.CustomerPOID = CustomerAddress.CustomerPOID");
  stmt.setLong(1, customerPOID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerId(rs.getLong("CustomerPOID"));
    customer.setName(rs.getString("Name"));
    customer.setPhoneNumber(rs.getString("PhoneNumber"));
    customer.setStreet(rs.getString("Street"));
    customer.setCity(rs.getString("City"));
    customer.setState(rs.getSring("State"));
    customer.setZipCode(rs.getString("ZipCode"));
  }
return customer;
}
```

Операция рефакторинга “Замена столбца”

Эта операция позволяет заменить существующий неключевой столбец новым столбцом (рис. 6.14).

Чтобы заменить столбец, который является частью ключа, либо первичного, либо альтернативного, необходимо воспользоваться операциями рефакторинга “Введение суррогатного ключа” (с. 123) и “Замена суррогатного ключа естественным ключом” (с. 168).

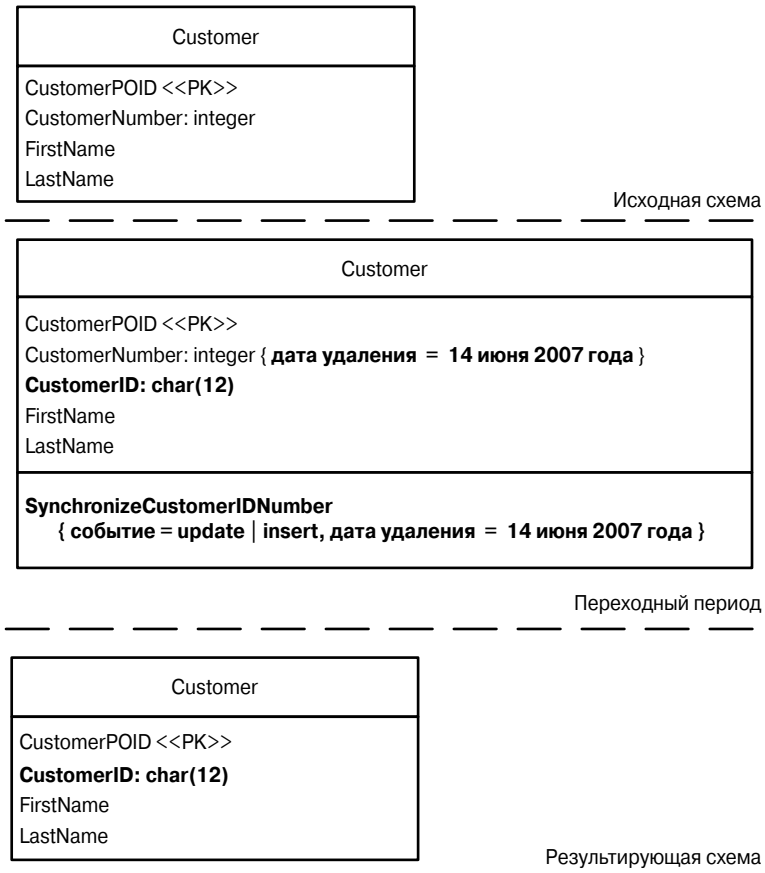


Рис. 6.14. Замена столбца `Customer.CustomerNumber`

Обоснование

Необходимость в использовании операции “Замена столбца” может быть обусловлена двумя причинами. Во-первых, может оказаться, что со временем изменилось назначение столбца. Чаще всего возникает такая ситуация, что по прошествии определенного времени столбец начинает использоваться для другой цели, вследствие чего возникает необходимость изменить его тип. Например, предположим, что до сих пор применялись числовые идентификаторы клиентов, а теперь участники разработки, отвечающие за экономическую часть проекта, решили перейти к использованию алфавитно-цифровых идентификаторов. Во-вторых, проведение данной операции рефакторинга может потребоваться в качестве промежуточного шага в направлении к реализации других операций рефакторинга. Еще одна распространенная причина замены существующего столбца состоит в том, что такая замена часто становится важным шагом на пути к слиянию двух аналогичных источников данных или подготовкой к применению операции “Осуществление стратегии консолидированных ключей” (с. 197), поскольку необходимо обеспечить согласованность по типу и формату с другим столбцом.

Потенциальные преимущества и недостатки

При замене столбца возникает существенный риск потери информации при переносе данных из исходного столбца в столбец, применяемый для замены. Такие предпосылки становятся особенно значимыми, если между типами двух столбцов имеются существенные различия, например, преобразование типа данных `CHAR` в тип данных `VARCHAR`, а также `NUMERIC` в `CHAR` является несложным, а преобразование типа данных `CHAR` в тип данных `NUMERIC` может быть связано с возникновением нарушений в работе, если исходный столбец содержит нечисловые символы.

Процедура обновления схемы

Чтобы обеспечить применение операции “Замена столбца”, необходимо выполнить описанные ниже действия.

1. **Ввести новый столбец.** Добавить столбец к целевой таблице с помощью команды `ADD COLUMN` языка SQL. На рис. 6.14 таковым является столбец `CustomerID`.
2. **Обозначить исходный столбец как устаревший.** Столбец `CustomerNumber` должен быть отмечен как предназначенный для удаления в конце выбранного переходного периода.
3. **Ввести триггер синхронизации.** Как показано на рис. 6.14, необходимо ввести в действие триггер, предназначенный для копирования данных из одного столбца в другой в течение переходного периода. Этот триггер должен вызываться при любом изменении в строке данных.
4. **Обновить другие таблицы.** Если столбец `CustomerNumber` используется в других таблицах как часть внешнего ключа, то может потребоваться заменить аналогичным образом и соответствующие столбцы внешнего ключа, а также обновить все относящиеся к ним определения индексов.

В следующем примере кода SQL приведены операторы DDL, предназначенные для замены столбца, создания триггера синхронизации и в конечном итоге для удаления столбца и триггера по истечении переходного периода:

```
ALTER TABLE Customer ADD CustomerID CHAR(12);

COMMENT ON Customer.CustomerID 'Replaces CustomerNumber column,
finaldate = 2007-06-14';

COMMENT ON Customer.CustomerNumber 'Replaced with CustomerID, dropdate
= 2007-06-14';


CREATE OR REPLACE TRIGGER SynchronizeCustomerIDNumber
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
  BEGIN
  IF :NEW.CustomerID IS NULL THEN
    :NEW.CustomerID:=
      formatCustomerNumber(:New.CustomerNumber);
  END IF;
  IF :NEW.CustomerNumber IS NULL THEN
    :New.CustomerNumber := :New.CustomerID;
  END IF;
END;
/

-- 14 июня 2007 года
DROP TRIGGER SynchronizeCustomerIDNumber;
ALTER TABLE Customer DROP COLUMN CustomerNumber;
```

Процедура переноса данных

Вначале следует скопировать данные из столбца `CustomerNumber` в столбец `CustomerID`, а затем поддерживать синхронизацию данных в течение переходного периода (например, с помощью хранимых процедур). Как было описано выше, при выполнении этого требования могут возникнуть проблемы, если форматы данных существенно отличаются друг от друга. Перед применением операции "Замена столбца" может обнаружиться, что необходимо применить одну или несколько операций рефакторинга качества данных для предварительной подготовки исходных данных. Код копирования значений в новый столбец показан ниже.

```
UPDATE Customer SET CustomerID = CustomerNumber;
```

Процедура обновления программ доступа

Основная проблема состоит в том, что должны быть подвергнуты рефакторингу внешние программы, чтобы они могли работать с новым типом данных и форматом `CustomerID`. Это может повлечь за собой необходимость подготовить код преобразования, позволяющий выполнять прямые и обратные преобразования данных из старого форма-

та в новый. А более дальновидная стратегия, хотя потенциально и более дорогостоящая, может состоять в том, чтобы код внешних программ был полностью переработан в целях использования нового формата данных. Ниже приведен фрагмент кода, в котором показано, как можно изменить имя столбца и тип данных в прикладном коде.

```
// Код до рефакторинга
public Customer findByCustomerID(Long customerID) {
  Customer customer = new Customer();
  stmt = DB.prepare("SELECT CustomerPOID, "+
    "CustomerNumber, FirstName, LastName " +
    "FROM Customer " +
    "WHERE CustomerPOID = ?");
  stmt.setLong(1, customerID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerPOID(rs.getLong("CustomerPOID"));
    customer.setCustomerNumber(rs.getInt("CustomerNumber"));
    customer.setFirstName(rs.getString("FirstName"));
    customer.setLastName(rs.getString("LastName"));
  }
return customer;
}

// Код после рефакторинга
public Customer findByCustomerID(Long customerID) {
  Customer customer = new Customer();
  stmt = DB.prepare("SELECT CustomerPOID, "+
    "CustomerID, FirstName, LastName " +
    "FROM Customer " +
    "WHERE CustomerPOID = ?");
  stmt.setLong(1, customerID);
  stmt.execute();
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    customer.setCustomerPOID(rs.getLong("CustomerPOID"));
    customer.setCustomerID(rs.getString("CustomerID"));
    customer.setFirstName(rs.getString("FirstName"));
    customer.setLastName(rs.getString("LastName"));
  }
return customer;
}
```

Операция рефакторинга “Замена связи “один ко многим” ассоциативной таблицей”

Эта операция позволяет заменить ассоциативной таблицей связь “один ко многим” между двумя таблицами (рис. 6.15).

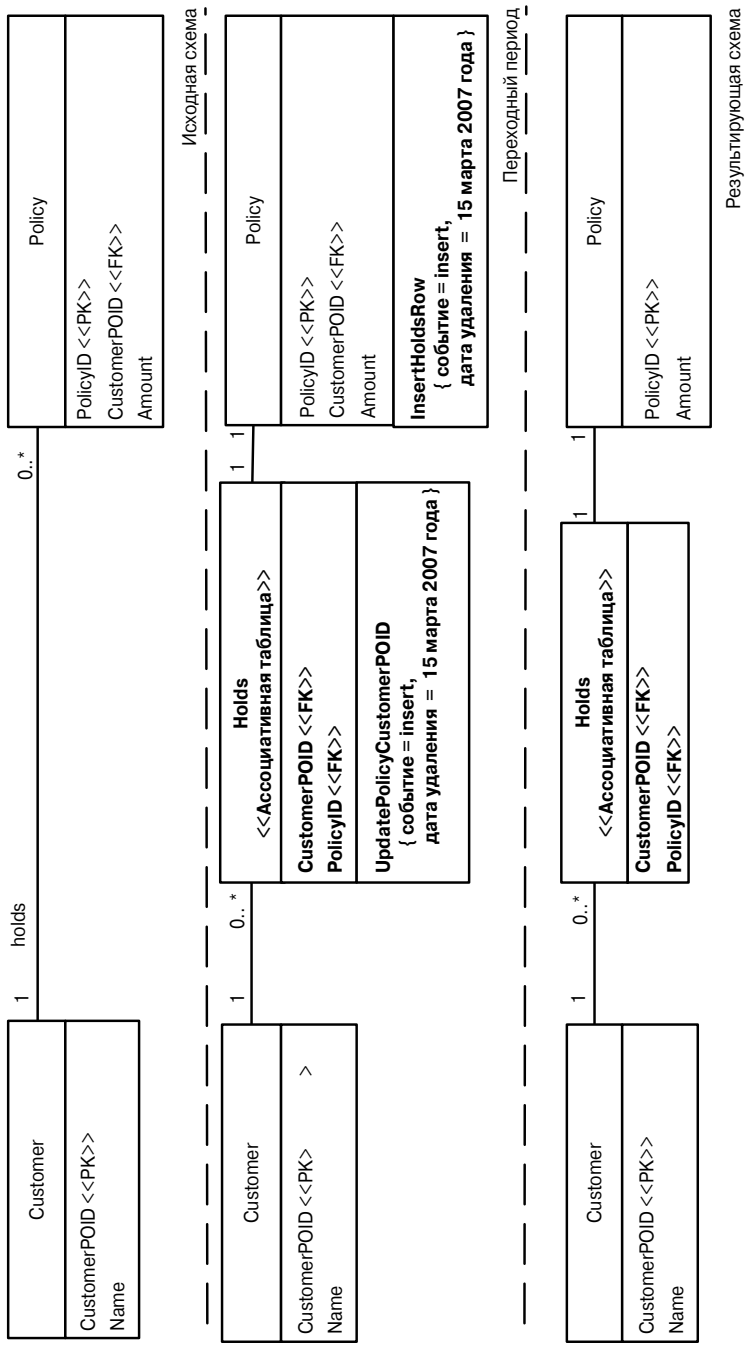


Рис. 6.15. Замена связи "один ко многим" ассоциативной таблицей

Обоснование

Основная причина введения ассоциативной таблицы между двумя таблицами может быть обусловлена необходимостью ввести в дальнейшем связь “многие ко многим” между этими таблицами. Ситуация, в которой связь “один ко многим” со временем развивается, преобразуясь в связь “многие ко многим”, встречается весьма часто. Например, предположим, что у любого конкретного сотрудника в текущий момент имеется самое большее один руководитель. (Президент компании является единственным лицом, не имеющим руководителя.) Но в компании задуман переход к матричной организационной структуре, в которой потенциально один сотрудник может отчитываться перед несколькими руководителями. Связь “один ко многим” является подмножеством связи “многие ко многим”, поэтому введение новой ассоциативной таблицы позволила бы отразить существующую иерархическую организационную структуру и вместе с тем подготовиться к будущей матричной структуре. Может также потребоваться ввести дополнительную информацию в саму связь, поскольку отсутствует возможность представить эту информацию в какой-либо из существующих таблиц.

Потенциальные преимущества и недостатки

Если для реализации связи “один ко многим” вводится ассоциативная таблица, то схема базы данных неоправданно усложняется. Если рассматриваемая связь, скорее всего, не разовьется в связь “многие ко многим”, то не рекомендуется принимать этот подход за основу. После ввода в действие ассоциативных таблиц увеличивается количество соединений, которые приходится выполнять для получения требуемых данных, что приводит к снижению производительности, к тому же схема базы данных становится более сложной для понимания.

Процедура обновления схемы

Чтобы подготовиться к применению операции “Замена связи “один ко многим” ассоциативной таблицей”, необходимо выполнить следующие действия.

1. **Добавить ассоциативную таблицу.** На рис. 6.15 таковой является `Holds`. Столбцы этой таблицы представляют собой комбинацию первичных ключей таблиц `Customer` и `Policy`. Следует отметить, что некоторые таблицы не обязательно могут иметь первичный ключ, хотя такая ситуация встречается редко, но если дело обстоит именно так, то может потребоваться применить операцию рефакторинга “Введение суррогатного ключа”.
2. **Обозначить исходный столбец как устаревший.** Непосредственная связь между таблицами `Policy` и `Customer` больше не поддерживается, поэтому столбец `Policy.CustomerPOID` должен быть отмечен как предназначенный для удаления в конце переходного периода; это связано с тем, что данный столбец в настоящее время используется для обеспечения связи “один ко многим” с таблицей `Customer`, но в дальнейшем не потребуется.
3. **Добавить новый индекс.** Необходимо ввести новый индекс для таблицы `Holds` с помощью операции рефакторинга “Введение индекса” (с. 271).

4. **Ввести триггеры синхронизации.** Для таблицы `Policy` потребуется триггер, который будет заполнять значения ключа в таблице `Holds`, если соответствующие значения еще не существуют; этот триггер должен использоваться в течение переходного периода. Аналогичным образом, для таблицы `Holds` также потребуется триггер, позволяющий обеспечить заполнение столбца `Policy.CustomerPOID` должным образом.

Ниже приведен код, предназначенный для добавления таблицы `Holds`, введения индекса на таблице `Holds`, добавления триггеров синхронизации и, наконец, для удаления старой схемы и триггеров.

```
CREATE TABLE Holds (
  CustomerPOID BIGINT,
  PolicyID INT,
);

CREATE INDEX HoldsIndex ON Holds
    (CustomerPOID, PolicyID);

CREATE OR REPLACE TRIGGER InsertHoldsRow
BEFORE INSERT OR UPDATE OR DELETE
ON Policy
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF updating THEN
    UpdateInsertHolds;
  END IF;
  IF inserting THEN
    CreateHolds;
  END IF;
  IF deleting THEN
  RemoveHolds;
  END IF;
END;
/

CREATE OR REPLACE TRIGGER UpdatePolicyCustomerPOID
BEFORE INSERT OR UPDATE OR DELETE
ON Holds
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
IF updating THEN
  UpdateInsertPolicy;
END IF;
IF inserting THEN
  CreatePolicy;
END IF;
IF deleting THEN
  RemovePolicy;
END IF;
```

```
END;
/

--15 марта 2007 года
DROP TRIGGER InsertHoldsRow;
DROP TRIGGER UpdatePolicyCustomerPOID;
ALTER TABLE customer
DROP COLUMN balance;
```

Определение соглашения об именовании

Для ассоциативных таблиц чаще всего применяются два разных соглашения об именовании: в первом из них таблице присваивается такое же имя, как и исходной связи, как было сделано в рассматриваемом примере, а во втором предусматривается конкатенация имен двух таблиц, что в данном примере привело бы к определению имени `CustomerPolicy`.

Процедура переноса данных

Ассоциативная таблица должны быть заполнена путем копирования значений из столбцов `Policy.CustomerPOID` и `Policy.PolicyID` в столбцы `Holds.CustomerPOID` и `Holds.PolicyID` соответственно. Это действие может быть выполнено с помощью простого сценария SQL, например, следующим образом:

```
INSERT INTO Holds (CustomerPOID,PolicyID)
  SELECT CustomerPOID,PolicyID FROM Policy
```

Процедура обновления программ доступа

Для обновления внешних программ необходимо выполнить действия, описанные ниже.

1. **Удалить программные структуры, с помощью которых выполняются операции обновления внешнего ключа.** Весь код, в котором присваиваются значения столбцу `Policy.CustomerPOID`, должен быть подвергнут рефакторингу, чтобы в нем для поддержания связи должным образом осуществлялась запись в таблицу `Holds`.
2. **Переопределить соединения.** Может оказаться так, что во многих внешних программах доступа определены соединения, в которых участвуют таблицы `Customer` и `Policy`, реализованные либо с помощью программных конструкций SQL, либо с помощью метаданных. Эти соединения должны быть подвергнуты рефакторингу, чтобы их можно было применять для работы с таблицей `Holds`.
3. **Переопределить операции выборки.** В некоторых внешних программах может быть предусмотрен переход по базе данных путем выборки одной или нескольких строк одновременно; при этом осуществляется выборка данных на основе значений ключа, а затем переход от таблицы `Policy` к таблице `Customer`. Эти операции выборки должны быть обновлены соответствующим образом.

В следующем примере кода показано, как откорректировать прикладной код, для того чтобы операция выборки данных теперь осуществлялась с помощью соединения, в котором используется ассоциативная таблица:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT Customer.CustomerPOID, Customer.Name, "+
  "Policy.PolicyID,Policy.Amount " +
  "FROM Customer, Policy" +
  "WHERE Customer.CustomerPOID = Policy.CustomerPOID "+
  "AND Customer.CustomerPOID = ? ");
stmt.setLong(1,customerPOID);
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
  "SELECT Customer.CustomerPOID, Customer.Name, "+
  "Policy.PolicyID,Policy.Amount " +
  "FROM Customer, Holds, Policy" +
  "WHERE Customer.CustomerPOID = Holds.CustomerPOID "+
  "AND Holds.PolicyId = Policy.PolicyId "+
  "AND Customer.CustomerPOID = ? ");
stmt.setLong(1,customerPOID);
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга “Замена суррогатного ключа естественным ключом”

Эта операция предназначена для замены суррогатного ключа существующим естественным ключом (рис. 6.16). Рассматриваемая операция рефакторинга является обратной по отношению к операции “Введение суррогатного ключа” (с. 123).

Обоснование

Как описано ниже, необходимость в использовании операции “Замена суррогатного ключа естественным ключом” может быть обусловлена несколькими причинами.

- **Сокращение издержек.** Замена суррогатного ключа существующим естественным ключом может привести к сокращению издержек, связанных с реализацией в структуре таблицы средств сопровождения дополнительного столбца (столбцов) суррогатного ключа.
- **Унификация стратегии сопровождения ключей.** Для обеспечения возможности проведения операции “Осуществление стратегии консолидированных ключей” (с. 197) может быть принято решение прежде всего заменить существующий суррогатный первичный ключ “официально признанным” естественным ключом.

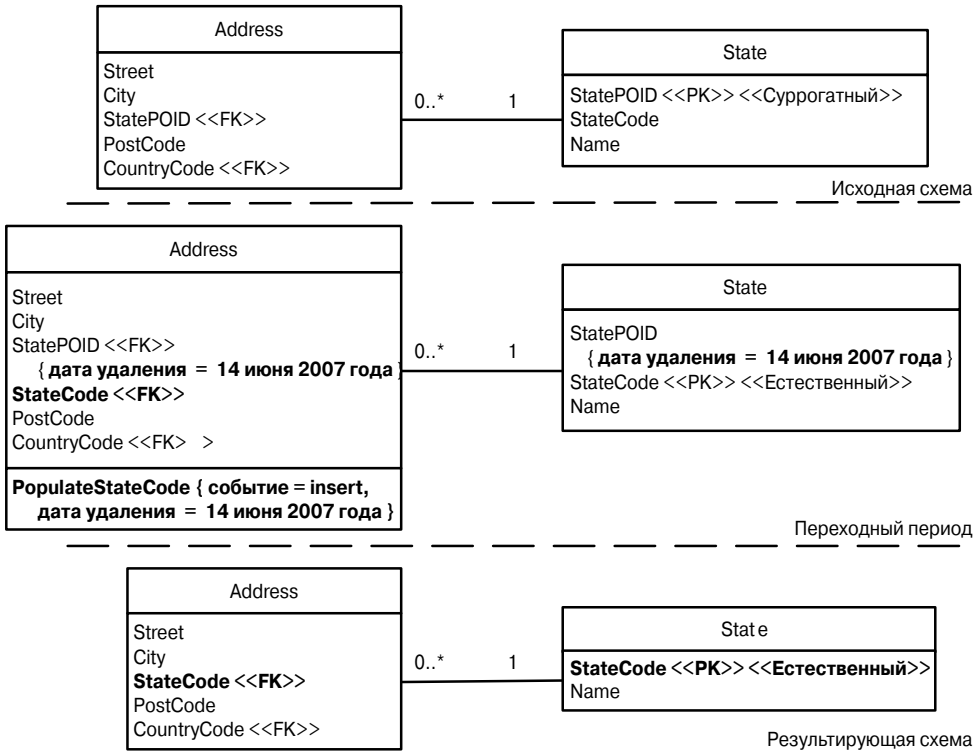


Рис. 6.16. Замена суррогатного ключа естественным ключом

- **Удаление ненужных ключей.** В процессе эксплуатации может быть обнаружено, что в таблицу был введен суррогатный ключ, который в действительности не требуется. Удаление неиспользуемых индексов всегда способствует повышению производительности.

Потенциальные преимущества и недостатки

Многие специалисты по базам данных расходятся во мнениях, когда речь идет об использовании суррогатных и естественных ключей, но реальность такова, что оба типа ключей имеют свою область применения. Если в базе данных применяются таблицы с естественными ключами, то для доступа к данным в каждой таблице в каждом внешнем приложении, а также в самой базе данных должен применяться отдельный уникальный способ. Дело в том, что иногда такой ключ может представлять собой отдельный числовой столбец, в ином случае — отдельный символьный столбец, а в других обстоятельствах — комбинацию из нескольких столбцов. Если же реализована стратегия согласованных суррогатных ключей, то доступ ко всем таблицам осуществляется точно в одинаковой форме, что позволяет упростить код. Таким образом, замена суррогатного ключа естественным ключом потенциально приводит к повышению сложности кода доступа к базе данных. А основное преимущество состоит в том, что упрощается схема таблицы.

Процедура обновления схемы

Поцедура осуществления операции "Замена суррогатного ключа естественным ключом" может оказаться сложной в силу того, что суррогатный ключ в принципе может участвовать в определенных связях. Дело в том, что суррогатный ключ является первичным ключом таблицы, поэтому велика вероятность того, что он образует также внешний ключ (или его часть) для других таблиц. В частности, необходимо выполнить описанные ниже действия.

1. **Определить столбец (столбцы), предназначенный для формирования нового первичного ключа.** На рис. 6.16 таковым является `StateCode`. (В состав первичного ключа могут входить не один, а несколько столбцов.) Столбец `StateCode` должен иметь уникальное значение в каждой строке, поскольку лишь в этом случае он может использоваться в качестве первичного ключа.
2. **Добавить новый индекс.** Если индекс еще не существует, то должен быть введен новый индекс для таблицы `State` на столбце `StateCode`.
3. **Обозначить исходный столбец как устаревший.** Столбец `StatePOID` должен быть отмечен как предназначенный для удаления в конце переходного периода.
4. **Обновить взаимосвязанные таблицы.** Если столбец `StatePOID` будет использоваться в других таблицах как часть внешнего ключа, то необходимо обновить эти таблицы, чтобы в них использовался новый ключ. Столбец (столбцы), который в настоящее время соответствует столбцу `StatePOID`, необходимо удалить с использованием операции "Удаление столбца" (с. 112). Необходимо также ввести новый столбец (столбцы), который соответствует столбцу `StateCode`, если этот столбец (столбцы) еще не существует. Требуется также обновить соответствующее определение (определения) индекса, чтобы в нем было отражено это изменение. Если столбец `StatePOID` используется во многих таблицах, то целесообразно рассмотреть возможность обновления этих таблиц по одной в целях упрощения работы.
5. **Обновить и, возможно, добавить триггеры ссылочной целостности.** Необходимо обновить все существующие триггеры, предназначенные для поддержки отношений ссылочной целостности между таблицами, чтобы они могли работать с соответствующими значениями `StateCode` в других таблицах.

На рис. 6.16 показано, как удалить суррогатный ключ `State.StatePOID` и заменить его существующим столбцом `State.StateCode`, который будет использоваться в качестве ключа. Для поддержки этого нового ключа необходимо добавить триггер `PopulateStateCode`, который вызывается при каждой вставке данных в таблицу `Address` и получает значение `State.StateCode`.

```
CREATE OR REPLACE TRIGGER PopulateStateCode
  BEFORE INSERT
  ON Address
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
```

```
    IF :NEW.StateCode IS NULL THEN
      :NEW.StateCode := getStatePOIDFromState(StatePOID);
    END IF;
  END;
/

ALTER TABLE Address ADD (CONSTRAINT AddressToStateForeignKey FOREIGN
KEY (StateCode) REFERENCES State;

-- 14 июня 2007 года
ALTER TABLE Address DROP CONSTRAINT AddressToStateForeignKey;
ALTER TABLE State DROP CONSTRAINT StatePrimaryKey;
ALTER TABLE State MODIFY StateCode NOT NULL;
ALTER TABLE State ADD CONSTRAINT StatePrimaryKey
      PRIMARY KEY (StateCode);
DROP TRIGGER PopulateStateCode;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных не требуется переносить какие-либо данные.

Процедура обновления программ доступа

Для обновления внешних программ доступа необходимо выполнить описанные ниже действия.

1. **Удалить код суррогатного ключа.** Код, применявшийся для присваивания значений столбцу суррогатного ключа (который мог быть реализован с помощью внешних приложений или базы данных) больше не должен вызываться. Этот код может даже вообще больше не потребоваться.
2. **Ввести в действие соединения, основанные на новом ключе.** Возможно, что во многих внешних программах доступа определены соединения, в которых участвует таблица `State`, реализованные с применением программных конструкций SQL или метаданных. Эти соединения должны быть подвергнуты рефакторингу, чтобы они могли работать со столбцом `StateCode`, а не `StatePOID`.
3. **Обеспечить выполнение операций выборки на основе нового ключа.** В некоторых внешних программах предусматривается прохождение по базе данных путем единовременного получения одной или нескольких строк; при этом выборка данных осуществляется на основе значений ключа. Эти операции выборки должны быть обновлены аналогичным образом.

В следующем Hibernate-отображении показано, как обеспечить, чтобы таблицы, указанные в ссылке, ссылались на новые ключи, и как исключить дальнейшее формирование значений в столбцах POID:

```
// Отображение до рефакторинга
<hibernate-mapping>
<class name="State" table="STATE">
  <id name="id" column="STATEPOID">
```

```
    <generator class="IdGenerator"/>
  </id>
  <property name="stateCode" />
  <property name="name" />
</class>
</hibernate-mapping>

<hibernate-mapping>
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESSID" >
    <generator class="IdGenerator"/>
  </id>
  <property name="streetLine" />
  <property name="city"  />
  <property name="postalCode"  />
  <many-to-one name="state" class="State"
    column="STATEPOID" not-null="true"/>
  <many-to-one name="country" class="Country"
    column="COUNTRYID" not-null="true"/>
</class>
</hibernate-mapping>

// Отображение после рефакторинга
<hibernate-mapping>
<class name="State" table="STATE">
  <property name="stateCode" />
  <property name="name" />
</class>
</hibernate-mapping>

<hibernate-mapping>
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESSID" >
    <generator class="IdGenerator"/>
  </id>
  <property name="streetLine" />
  <property name="city"  />
  <property name="postalCode"  />
  <many-to-one name="state" class="State"
    column="STATECODE" not-null="true"/>
  <many-to-one name="country" class="Country"
    column="COUNTRYID" not-null="true"/>
</class>
</hibernate-mapping>
```

Операция рефакторинга “Разбиение столбца”

Эта операция применяется для разбиения столбца на один или несколько столбцов, находящихся в одной и той же таблице (рис. 6.17).

Следует отметить, что если один или несколько новых столбцов должны появиться в другой таблице, то сначала необходимо применить операцию “Разбиение столбца”, а затем — “Перемещение столбца” (с. 139).

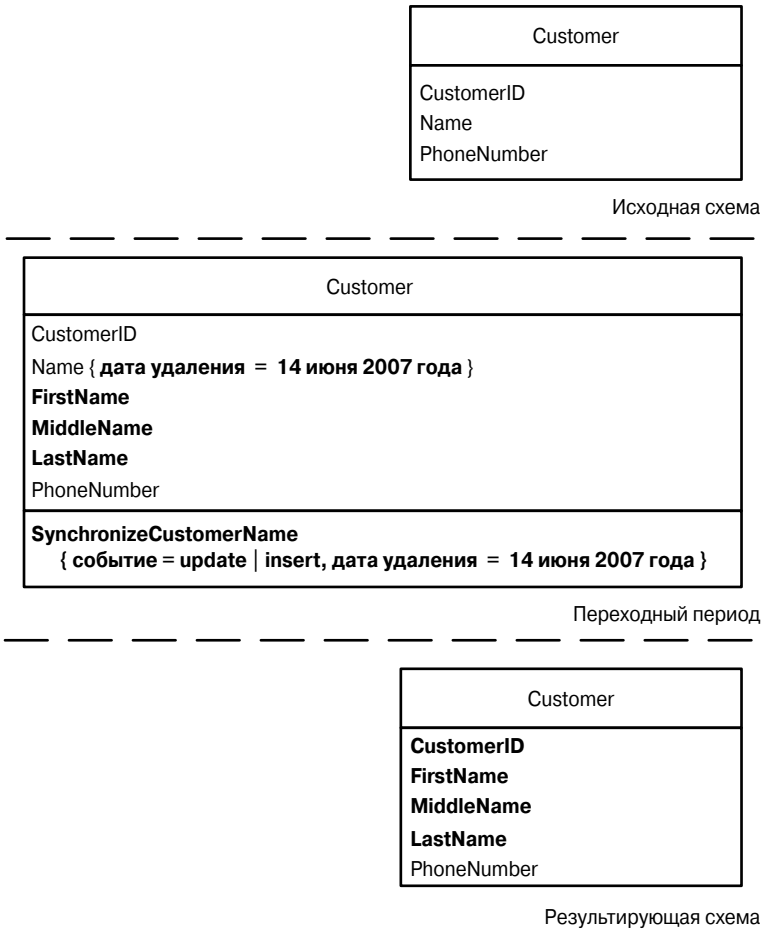


Рис. 6.17. Разбиение столбца `Customer.Name`

Обоснование

Необходимость в использовании операции “Разбиение столбца” может быть обусловлена двумя причинами. Во-первых, может потребоваться более высокая степень детализации данных. Например, предположим, что в таблице `Customer` имеется столбец `Name`, который содержит полное имя лица, но этот столбец необходимо разбить так, чтобы данные об имени, отчестве и фамилии хранились в виде отдельных столбцов, `FirstName`, `MiddleName` и `LastName`. Во-вторых, может оказаться так, что один столбец используется для нескольких назначений. Предположим, что исходный столбец был введен для отслеживания состояния счета `Account`, а теперь он используется также для отслеживания типа счета `Account`. В частности, столбец `Account.Status` содержит данные о состоянии счета (такие как “Открыт” (`Open`), “Закрыт” (`Closed`), “По счету выполнен овердрафт” (`OverDrawn`) и т.д.). Но, не учитывая истинного назначения этого столбца, какие-то пользователи стали приме-

нять его для хранения информации о типе счета, такой как "Текущий" (Checking), "Сберегательный" (Savings) и т.д. Необходимо разнести эти данные по отдельным полям, чтобы избежать появления программных ошибок, связанных с использованием одного и того же столбца в двух разных целях.

Потенциальные преимущества и недостатки

Выполнение этой операции рефакторинга базы данных может привести к тому, что после разделения столбцов появятся дубликаты данных. Осуществляя разбиение столбца, который (по вашему мнению) используется одновременно в различных целях, вы подвергаетесь риску того, что фактически введете новые столбцы для применения с одним и тем же назначением. (В таком случае может быть обнаружено, что необходимо применить операцию "Слияние столбцов".) Решение о том, следует ли выполнять разбиение столбца, должно приниматься с учетом реального использования этого столбца.

Процедура обновления схемы

Для выполнения операции "Разбиение столбца" необходимо вначале ввести новые столбцы. Добавьте столбец в таблицу с помощью команды ADD COLUMN языка SQL. На рис. 6.17 таковыми являются стобцы FirstName, MiddleName и LastName. Этот шаг необязателен, поскольку может оказаться так, что данные, полученные в результате разбиения, относятся к одному из существующих столбцов. Затем необходимо ввести триггер синхронизации для обеспечения того, чтобы столбцы оставались синхронизированными друг с другом. Этот триггер должен вызываться при внесении любых изменений в столбцы.

На рис. 6.17 приведен пример, в котором имя лица первоначально хранится в таблице Customer в столбце Name. Но со временем было обнаружено, что полное имя требуется лишь для немногих приложений, а в большинстве других приложений используются компоненты полного имени, особенно фамилия клиента. Кроме того, было обнаружено, что во многих приложениях дублируется код разбиения значений столбца Name, что является потенциальным источником программных ошибок. Поэтому было решено разбить один столбец Name на три столбца, FirstName, MiddleName и LastName, которые отражают фактический характер использования данных. Для поддержки синхронизации значений в столбцах введен триггер SynchronizeCustomerName. Изменения в схеме реализованы в следующем коде:

```
ALTER TABLE Customer ADD FirstName VARCHAR(40);

COMMENT ON Customer.FirstName 'Added as the result of splitting
Customer.Name finaldate = December 14 2007';

ALTER TABLE Customer ADD MiddleName VARCHAR(40);

COMMENT ON Customer.MiddleName 'Added as the result of splitting
Customer.Name finaldate = December 14 2007';

ALTER TABLE Customer ADD LastName VARCHAR(40);

COMMENT ON Customer.LastName 'Added as the result of splitting
```

```
Customer.Name finaldate = December 14 2007';

COMMENT ON Customer.Name 'Replaced with FirstName, LastName and
MiddleName, will be dropped December 14 2007';

-- Триггер, обеспечивающий синхронизацию всех разделенных столбцов
CREATE OR REPLACE TRIGGER SynchronizeCustomerName
  BEFORE INSERT OR UPDATE
  ON Customer
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF :NEW.FirstName IS NULL THEN
      :NEW.FirstName := getFirstName(Name);
    END IF;
    IF :NEW.MiddleName IS NULL THEN
      :NEW.MiddleName := getMiddleName(Name);
    END IF;
    IF :NEW.LastName IS NULL THEN
      :NEW.LastName := getLastName(Name);
    END IF;
END;
/


-- 14 декабря 2007 года
ALTER TABLE Customer DROP COLUMN Name;
DROP TRIGGER SynchronizeCustomerName;
```

Процедура переноса данных

Необходимо скопировать все данные из исходного столбца (столбцов) в столбцы, созданные в результате разбиения, в данном случае из столбца `Customer.Name` в столбцы `FirstName`, `MiddleName` и `LastName`. Операторы DML, предназначенные для первоначального разбиения данных столбца `Name` и перемещения их в три новых столбца, показаны в следующем коде. (В целях сокращения исходный код трех хранимых функций, которые вызываются в этом примере, не показан.)

```
/* Одноразовый перенос данных из таблицы Customer.Name в таблицы
Customer.FirstName, Customer.MiddleName и Customer.LastName. Если оба
набора столбцов активизированы, необходимо предусмотреть триггер,
который обеспечивает синхронизацию обоих наборов столбцов */

UPDATE Customer SET FirstName = getFirstName(Name);
UPDATE Customer SET MiddleName = getMiddleName(Name);
UPDATE Customer SET LastName = getLastName(Name);
```

Процедура обновления программ доступа

Необходимо тщательно проанализировать используемые программы доступа, а затем обновить их должным образом в течение переходного периода. Кроме очевидных обновлений, связанных с необходимостью обеспечить работу со столбцами `FirstName`, `Mid-`

dleName и LastName, а не с прежним столбцом, может также потребоваться осуществить следующие действия.

1. **Удалить код разбиения.** Может оказаться так, что существует код, применяемый для разбиения данных исходных столбцов на атрибуты данных, аналогичные тем, которые будут находиться в столбцах, созданных в результате разбиения. Этот код должен быть подвергнут рефакторингу и, возможно, полностью удален.
2. **Обновить код проверки данных для работы с данными, полученными в результате разбиения.** Может применяться некоторый код проверки данных, необходимость в использовании которого обусловлена исключительно тем, что не было выполнено разбиение столбцов. Например, если в столбце Customer.Name хранится полное имя, то может требоваться код проверки, позволяющий убедиться в том, что в составе полного имени указаны имя и фамилия. После разбиения столбца необходимость в использовании такого кода проверки может быть исключена.
3. **Подвергнуть рефакторингу пользовательский интерфейс.** После разбиения исходного столбца может потребоваться в случае необходимости использовать на уровне представления данные более мелко детализированные данные, если это еще не сделано.

В следующем коде показано, как обеспечить использование в приложении доступных ему данных, имеющих более высокую степень детализации:

```
// Код до рефакторинга
public Customer findByCustomerID(Long customerID)  {
Customer customer = new Customer();
stmt = DB.prepare("SELECT CustomerID, "+
"Name, PhoneNumber " +
"FROM Customer " +
"WHERE CustomerID = ?");
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
 customer.setCustomerId(rs.getLong("CustomerID"));
 String name = rs.getString("Name");
 customer.setFirstName(getFirstName(name));
 customer.setMiddleName(getMiddleName(name));
 customer.setLastName(getMiddleName(name));
 customer.setPhoneNumber(rs.getString("PhoneNumber"));
}
return customer;
}

// Код после рефакторинга
public Customer findByCustomerID(Long customerID)  {
Customer customer = new Customer();
stmt = DB.prepare("SELECT CustomerID, "+
"FirstName, MiddleName, LastName, PhoneNumber " +
"FROM Customer " +
"WHERE CustomerID = ?");
```

```
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
  customer.setCustomerId(rs.getLong("CustomerID"));
  customer.setFirstName(rs.getString("FirstName"));
  customer.setMiddleName(rs.getString("MiddleName"));
  customer.setLastName(rs.getString("LastName"));
  customer.setPhoneNumber(rs.getString("PhoneNumber"));
}
return customer;
}
```

Операция рефакторинга “Разбиение таблицы”

Эта операция позволяет выполнить разбиение существующей таблицы по вертикали (например, по столбцам) на одну или несколько таблиц (рис. 6.18).

Следует отметить, что если местом назначения столбцов, полученных в результате разбиения, фактически становится существующая таблица, то в действительности применяется операция “Перемещение столбца” (с. 139). Для разбиения таблицы по горизонтали (например, по строкам) следует применять операцию “Перемещение данных” (с. 219).

Обоснование

Необходимость в использовании операции “Разбиение таблицы” может быть обусловлена несколькими причинами, описанными ниже.

- **Повышение производительности.** Очень часто для большинства приложений требуется основная коллекция атрибутов данных, относящихся к каждой конкретной сущности, а также определенное подмножество неосновных атрибутов данных. Например, предположим, что в состав основных столбцов таблицы `Employee` входят столбцы, необходимые для хранения имени, адреса и номера телефона сотрудника, а неосновные столбцы включают столбец с фотографией, `Picture`, а также информацию о зарплате. Безусловно, объем графических данных в столбце `Employee.Picture` весьма велик, но фотографии требуются лишь в нескольких приложениях, поэтому целесообразно предусмотреть возможность выделения этих данных в отдельную таблицу. Это позволяет повысить быстродействие операций доступа при выборке данных для тех приложений, для которых требуются все столбцы из таблицы `Employee`, но не нужна фотография.
- **Ограничение доступа к данным.** Может потребоваться ограничить доступ к некоторым столбцам, возможно, к информации о зарплате, хранящейся в таблице `Employee`. Для этого достаточно выполнить разбиение таблицы, перенести соответствующие столбцы в отдельную таблицу и распространить на эту таблицу правила управления защитой доступа (Security Access Control — SAC).
- **Уменьшение количества повторяющихся групп данных (применение первой нормальной формы).** Может оказаться так, что исходная таблица была спроектирована в то время, как требования еще не были до конца сформулированы, или ее проектированием занимались люди, не понимающие, для чего необходимо нормализовать

структуры данных [**4**; **15**]. Например, предположим, что в таблице `Employee` хранятся описания результатов пяти предыдущих аттестаций каждого конкретного сотрудника. Эта информация представляет собой повторяющуюся группу, которую необходимо вынести в отдельную таблицу `EmployeeEvaluation`, выделив ее путем разбиения.

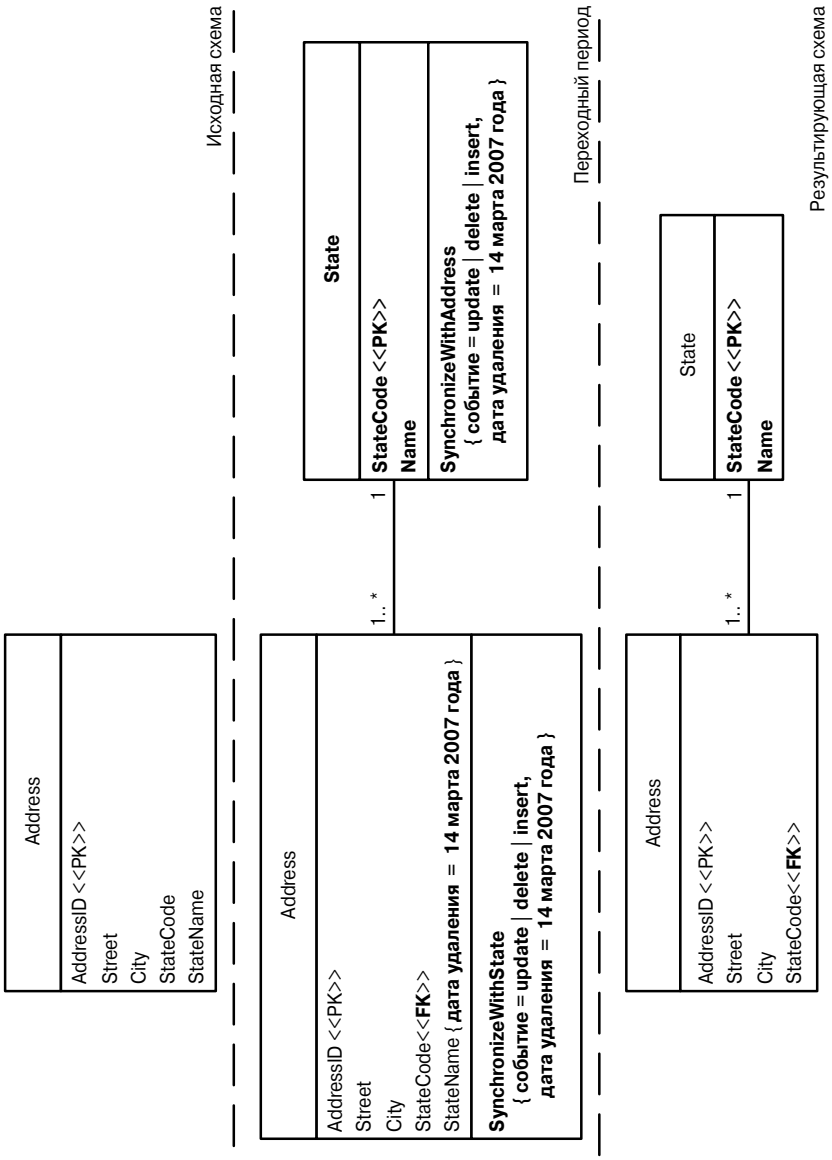


Рис. 6.18. Разбиение таблицы `Address`

Потенциальные преимущества и недостатки

Осуществляя разбиение таблицы, которая (по вашему мнению) используется одновременно в различных целях, вы рискуете ввести новые таблицы для применения с одним и тем же назначением; в таком случае может быть обнаружено, что необходимо применить операцию “Слияние таблиц” (с. 133). Решение о том, следует ли выполнять разбиение таблицы, должно приниматься с учетом реального использования этой таблицы.

Процедура обновления схемы

Для выполнения операции “Разбиение таблицы” необходимо вначале с помощью команды `CREATE TABLE` языка SQL создать необходимую таблицу (таблицы). Этот шаг необязателен, поскольку может оказаться так, что данные, полученные в результате разбиения, относятся к одной или нескольким существующим таблицам. В этой ситуации необходимо будет неоднократно применить операцию рефакторинга “Перемещение столбца” (с. 139). Кроме того, требуется ввести триггер, позволяющий обеспечить синхронизацию столбцов друг с другом. Триггер должен вызываться при любом изменении в таблице и должен быть реализован так, чтобы не возникали циклы.

На рис. 6.18 приведен пример, в котором информация адреса наряду с кодом и названием штата первоначально хранится в таблице `Address`. Чтобы уменьшить дублирование данных, было решено разбить таблицу `Address` на две таблицы, `Address` и `State`, что соответствует выполняемому в настоящее время рефакторингу проекта таблицы. Кроме того, введены триггеры `SynchronizeWithAddress` и `SynchronizeWithState` для обеспечения синхронизации значений в таблицах:

```
CREATE TABLE State (
  StateCode VARCHAR(2) NOT NULL,
  Name VARCHAR(40) NOT NULL,
  CONSTRAINT PKState
    PRIMARY KEY (StateCode)
);

COMMENT ON State.Name 'Added as the result of splitting Address into
Address and State  drop date = December 14 2007';

-- Триггер, обеспечивающий синхронизацию всех разделенных таблиц
CREATE OR REPLACE TRIGGER SynchronizeWithAddress
BEFORE INSERT OR UPDATE
ON Address
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF updating THEN
    FindOrCreateState;
  END IF;
  IF inserting THEN
    CreateState;
  END IF;
END;
/
```

```
CREATE OR REPLACE TRIGGER SynchronizeWithState
BEFORE UPDATE OF Statename
ON State
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF updating THEN
    FindAndUpdateAllAddressesForStateName;
  END IF;
END;
/

-- 14 декабря 2007 года
ALTER TABLE Address DROP COLUMN StateName;
DROP TRIGGER SynchronizeWithAddress;
DROP TRIGGER SynchronizeWithState;
```

Процедура переноса данных

Необходимо скопировать все данные из исходного столбца (столбцов) в столбцы новой таблицы. В примере, показанном на рис. 6.18, эта операция копирования должна предусматривать перенос данных из столбцов `Address.StateCode` и `Address.StateName` соответственно в столбцы `State.StateCode` и `State.Name`. В следующем коде показано, как выполнить первоначально такой перенос данных:

```
/* Одноразовый перенос данных из столбцов Address.StateCode и
Address.StateName в таблицу State. Если оба набора столбцов активны,
необходимо предусмотреть триггер, который обеспечивает синхронизацию
обоих наборов столбцов */

INSERT INTO State (StateCode, Name)
  SELECT StateCode,StateName FROM Address
  WHERE StateCode IS NOT NULL AND StateName IS NOT NULL
  GROUP BY StateCode, StateName;
```

Процедура обновления программ доступа

Необходимо тщательно проанализировать программы доступа, а затем обновить их должным образом в течение переходного периода. Кроме очевидных обновлений, связанных с необходимостью обеспечить работу с новыми столбцами, а не с прежними, может также потребоваться провести некоторые обновления.

1. **Ввести новые метаданные таблицы.** Если используется инфраструктура обеспечения перманентности, основанная на метаданных, то необходимо ввести новые метаданные для таблицы `State` и откорректировать метаданные, относящиеся к таблице `Address`.
2. **Обновить код SQL.** Аналогичным образом, должен быть обновлен весь внедренный код SQL, получающий доступ к таблице `Address`, для того, чтобы в нем случае необходимости выполнялись соединения с таблицей `State`.

Это может привести к небольшому снижению производительности рассматриваемого кода.

3. **Подвергнуть рефакторингу пользовательский интерфейс.** После разбиения исходной таблицы может потребоваться использовать на уровне представления более мелко детализированные данные, если это еще не сделано.

В следующем Hibernate-отображении показано, как выполнить разбиение таблицы `Address` и создать новую таблицу `State`:

```
// Отображение до рефакторинга
<hibernate-mapping>
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESSID">
    <generator class="IdGenerator"/>
  </id>
  <property name="street" />
  <property name="city" />
  <property name="stateCode" />
  <property name="stateName" />
</class>
</hibernate-mapping>

// Отображение после рефакторинга
// Таблица Address
<hibernate-mapping>
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESSID">
    <generator class="IdGenerator"/>
  </id>
  <property name="street" />
  <property name="city" />
  <many-to-one name="state" class="State"
    column="STATECODE" not-null="true"/>
</class>
</hibernate-mapping>

// Таблица State
<hibernate-mapping>
<class name="State" table="STATE">
  <id name="stateCode" column="stateCode">
    <generator class="assigned"/>
  </id>
  <property name="stateName" />
</class>
</hibernate-mapping>
```

Глава 7

Операции рефакторинга качества данных

Операции рефакторинга качества данных представляют собой изменения, которые способствуют улучшению качества информации, содержащейся в базе данных. Операции рефакторинга качества данных повышают и (или) гарантируют согласованность и применимость значений данных, хранящихся в базе данных. Операции рефакторинга качества данных перечислены ниже.

- Операция рефакторинга “Добавление поисковой таблицы”.
- Операция рефакторинга “Применение стандартных кодовых обозначений”.
- Операция рефакторинга “Применение стандартного типа”.
- Операция рефакторинга “Осуществление стратегии консолидированных ключей”.
- Операция рефакторинга “Уничтожение ограничения столбца”.
- Операция рефакторинга “Уничтожение значения, заданного по умолчанию”.
- Операция рефакторинга “Уничтожение столбца, не допускающего NULL-значений”.
- Операция рефакторинга “Введение ограничения столбца”.
- Операция рефакторинга “Введение общего формата”.
- Операция рефакторинга “Введение заданного по умолчанию значения”.
- Операция рефакторинга “Преобразование столбца в недопускающий NULL-значения”.
- Операция рефакторинга “Перемещение данных”.
- Операция рефакторинга “Замена кодового обозначения типа флажками свойств”.

Проблемы, часто возникающие при осуществлении операций рефакторинга качества данных

Операции рефакторинга качества данных изменяют значения данных, хранящиеся в базе данных, поэтому в результате выполнения всех этих операций могут возникать некоторые распространенные проблемы. В связи с этим приходится предпринимать описанные ниже действия.

1. **Исправление нарушенных ограничений.** Может оказаться так, что на данных, затронутых изменениями, определены ограничения. В таком случае можно применить операцию "Уничтожение ограничения столбца" (с. 201), чтобы сначала удалить ограничение, а затем применить операцию "Введение ограничения столбца" (с. 207) для повторного введения в действие ограничения, отражающего значения улучшенных данных.
2. **Исправление нарушенных представлений.** Представления часто содержат ссылки на заданные в виде программных конструкций значения данных в предложениях `WHERE`; при этом такие данные часто служат критерием выборки подмножества данных. В результате этого после изменения значений данных функционирование таких представлений может быть нарушено. Необходимо найти такие нарушенные представления, выполнив тестовый набор; кроме того, следует провести поиск определений представлений, которые ссылаются на столбцы, хранящие данные, подвергнутые изменениям.
3. **Исправление нарушенных хранимых процедур.** Со значениями улучшенных данных могут быть потенциально связаны переменные, которые определены в хранимых процедурах, любые передаваемые им параметры, возвращаемые значения, вычисленные с помощью хранимых процедур, а также любой определенный в этих процедурах код SQL. Следует стремиться к тому, чтобы применяемые тесты позволяли вскрыть все проблемы, связанные с нарушением функционирования бизнес-логики, которые являются результатом применения любых операций рефакторинга качества данных; в противном случае приходится проводить поиск всего кода хранимых процедур, в котором осуществляется доступ к столбцу (столбцам), хранящему данные, подвергшиеся изменениям.
4. **Обновление данных.** По всей вероятности, на время обновления придется заблокировать исходные данные, подвергающиеся обновлению, что может отрицательно повлиять на производительность и доступность данных для приложения (приложений). При этом можно применить одну из двух стратегий. Во-первых, можно заблокировать все данные и в это время выполнить все обновления. Во-вторых, можно блокировать одновременно лишь подмножества данных, возможно, только отдельные строки, и обновлять только эти подмножества. Первый подход гарантирует согласованность, но связан с риском снижения производительности при обработке больших объемов данных, поскольку обновление миллионов строк может потребовать много времени и в этот период будет исключена возможность вносить обновления с помощью обычных приложений. Второй подход позволяет обеспечить работу приложений с

исходными данными в процессе обновления, но связан с риском появления несовместимостей между строками, поскольку в некоторых строках будут находиться более старые, “низкокачественные” значения, а в других строках данные будут уже обновлены.

Операция рефакторинга “Добавление поисковой таблицы”

Эта операция позволяет создать поисковую таблицу для существующего столбца (рис. 7.1).

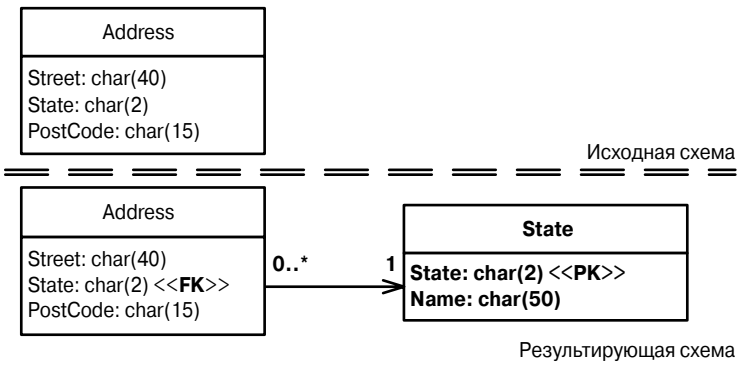


Рис. 7.1. Добавление поисковой таблицы `State`

Обоснование

Необходимость в выполнении операции “Добавление поисковой таблицы” может быть обусловлена несколькими описанными ниже причинами.

- **Введение в действие средств поддержки ссылочной целостности.** В качестве примера можно указать, что ограничение ссылочной целостности может быть введено на существующем столбце `Address.State` для обеспечения качества данных.
- **Обеспечение поиска кода.** На практике часто возникает необходимость ввести определенный перечень кодов в базу данных, а не задавать эти коды в виде перечисления в каждом приложении. Поисковая таблица часто кэшируется в памяти.
- **Замена ограничения столбца.** В таком случае после ввода в таблицу столбца добавляется ограничение столбца, позволяющее гарантировать постоянное наличие в этом столбце небольшого количества правильных кодовых обозначений. Но может оказаться так, что по мере развития приложения (приложений) потребуется все больше и больше кодовых обозначений, до тех пор, пока не наступит время, после которого будет проще сопровождать эти обозначения в поисковой таблице, а не обновлять ограничение столбца.
- **Предоставление подробных описаний.** Кроме определения допустимых кодовых обозначений, может также потребоваться хранить описательную информацию, относящуюся к этим кодовым обозначениям. Например, может оказаться так, что в таблице `State` для обозначения `CA` необходимо будет указать, что оно относится к штату Калифорния.

Потенциальные преимущества и недостатки

При вводе в действие поисковой таблицы необходимо рассмотреть два описанных ниже вопроса. Первым из них является заполнение данными; необходимо иметь возможность предоставить достоверные данные для заполнения поисковой таблицы. На практике это требование может показаться легко выполнимым, но из него следует, что необходимо руководствоваться определенным соглашением, касающимся семантики существующих значений данных, как в рассматриваемом примере столбца `Address.State`, который показан на рис. 7.1. Это проще сказать, чем сделать. Например, что касается ввода в действие поисковой таблицы `State`, то в некоторых приложениях могут использоваться данные обо всех 50 штатах США, а в других приложениях могут также рассматриваться 4 территории (Пуэрто-Рико, Гуам, округ Колумбия и Виргинские острова, принадлежащие США). В подобной ситуации можно либо ввести две поисковые таблицы, одну для 50 штатов, а другую для территорий, либо создать единственную таблицу, а затем ввести соответствующее программное средство проверки в приложения, для которых требуется лишь подмножество этих справочных данных.

Вторая проблема состоит в том, что ввод в действие ограничения внешнего ключа может оказать отрицательное влияние на производительность. (Дополнительные сведения по этой теме приведены в описании операции рефакторинга "Добавление ограничения внешнего ключа" на с. 244.)

Процедура обновления схемы

Как показано на рис. 7.1, для обновления схемы базы данных необходимо выполнить следующие действия.

1. **Определить структуру таблицы.** Необходимо уточнить, какой столбец (столбцы) должен войти в поисковую таблицу (`State`).
2. **Создать таблицу.** В базе данных с помощью команды `CREATE TABLE` создается таблица `State`.
3. **Определить поисковые данные.** Необходимо определить, какие строки должны быть вставлены в таблицу `State`. Для этого может применяться операция рефакторинга "Вставка данных" (с. 317).
4. **Ввести ограничение ссылочной целостности.** Чтобы распространить действие ограничений ссылочной целостности со столбца с кодовыми обозначениями в исходной таблице (таблицах) на таблицу `State`, необходимо применить операцию рефакторинга "Добавление ограничения внешнего ключа" (с. 229).

В следующем примере кода приведены операторы DDL, позволяющие создать таблицу `State` и ввести ограничение внешнего ключа между этой таблицей и таблицей `Address`:

```
-- Создание поисковой таблицы
CREATE TABLE State (
  State CHAR(2) NOT NULL,
  Name CHAR(50),
  CONSTRAINT PKState
    PRIMARY KEY (State)
);
```

```
-- Введение внешнего ключа к поисковой таблице
ALTER TABLE Address ADD CONSTRAINT FK_Address_State
  FOREIGN KEY (State) REFERENCES State;
```

Процедура переноса данных

Необходимо обеспечить, чтобы значения данных в столбце `Address.State` имели соответствующие значения в таблице `State`. Наиболее простой способ заполнения столбца `State.State` состоит в копировании уникальных значений из столбца `Address.State`. Такой подход является в значительной степени автоматизированным, но при его использовании следует помнить, что полученные в результате строки должны быть проверены, чтобы можно было гарантировать отсутствие случайно введенных недопустимых значений данных; в случае обнаружения таких значений необходимо обновить должным образом и таблицу `Address`, и таблицу `State`. Если же предусмотрены столбцы с описательной информацией, такие как `State.Name`, то необходимо ввести в них соответствующие значения; такие действия часто осуществляются вручную, с помощью сценария или программы администрирования данных.

В следующем коде приведены операторы DDL, позволяющие заполнить таблицу `State` уникальными значениями, полученными из столбца `Address.State`. После этого выполняется унификация данных, в рассматриваемом случае для того, чтобы во всех адресах использовалось кодовое обозначение `TX`, а не `Tx`, `tx` или `Texas`. На заключительном этапе необходимо ввести названия штатов, соответствующие каждому коду штата. (В этом примере показано заполнение значений, относящихся только к трем штатам.)

```
-- Заполнение поисковой таблицы данными
INSERT INTO State (State)
  SELECT DISTINCT UPPER(State) FROM Address;

-- Обновление столбца Address.StateCode допустимыми значениями
-- и очистка данных

UPDATE Address SET State = 'TX' WHERE UPPER(State) ='TX';
...
-- Предоставление имен штатов
UPDATE State SET Name = 'Florida' WHERE State='FL';
UPDATE State SET Name = 'Illinois' WHERE State='IL';
UPDATE State SET Name = 'California' WHERE State='CA';
```

Процедура обновления программ доступа

После ввода в действие поисковой таблицы `State` необходимо добиться того, чтобы в дальнейшем во всех внешних программах использовались значения данных из этой таблицы. В следующем коде показано, как теперь можно получить название штата из таблицы `State` во внешней программе; до сих пор в таких программах указанная информация поступала из коллекции, реализованной с помощью программных конструкций в самой программе:

```
// Код после рефакторинга
ResultSet rs = statement.executeQuery(
  "SELECT State, Name FROM State");
```

В некоторых программах может быть предусмотрено кэширование этих значений данных, а в других — осуществление доступа к значениям таблицы `State` по мере необходимости; кэширование выполняется успешно, поскольку значения в таблице `State` изменяются редко. Кроме того, если наряду с операцией `Lookup Table` будет также введено ограничение внешнего ключа, то во внешних программах необходимо обеспечить обработку любых исключений, активизированных в базе данных. Дополнительные сведения на эту тему приведены в описании операции рефакторинга “Добавление ограничения внешнего ключа” (с. 229).

Операция рефакторинга “Применение стандартных кодовых обозначений”

Эта операция предусматривает применение стандартного набора кодовых обозначений к отдельному столбцу для обеспечения соответствия хранящихся в нем данных значениям аналогичных столбцов, хранящихся в другом месте в базе данных (рис. 7.2).

Обоснование

Проведение операции “Применение стандартных кодовых обозначений” может потребоваться в связи с осуществлением описанных ниже действий.

- **Унификация данных.** Если в базе данных применяются различные кодовые обозначения, имеющие одинаковое семантическое значение, то, как правило, лучше стандартизировать эти обозначения, чтобы иметь возможность использовать стандартные программные средства для обработки всех атрибутов данных. Например, если в столбце `Country.Code` применяется кодовое обозначение `USA`, а в столбце `Address.CountryCode` предусмотрено обозначение `US`, то появляются предпосылки возникновения нарушений в работе, поскольку исключена возможность правильно выполнить операцию соединения таблиц по этим двум столбцам. Поэтому во всей базе данных должно применяться какое-то согласованное значение, либо то, либо другое.
- **Поддержка ссылочной целостности.** Если возникнет необходимость применить операцию “Добавление ограничения внешнего ключа” (с. 229) к таблицам с учетом столбца с кодовыми обозначениями, то вначале следует стандартизировать эти кодовые обозначения.
- **Введение поисковой таблицы.** Если возникает необходимость применить операцию “Добавление поисковой таблицы” (с. 185), то при этом часто приходится заниматься стандартизацией кодовых обозначений, на основе которых происходит поиск.
- **Обеспечение соответствия корпоративным стандартам.** Во многих организациях приняты детализированные стандарты данных и моделирования данных, которые обязаны соблюдать группы разработчиков. Часто возникает такая ситуация, что при выполнении операции “Использование официально заданного источника данных” (с. 292) обнаруживается, что существующая в настоящее время схема данных не соответствует стандартам организации и поэтому должна быть подвергнута рефакторингу, чтобы в ней были отражены официально принятые кодовые обозначения для источника данных.

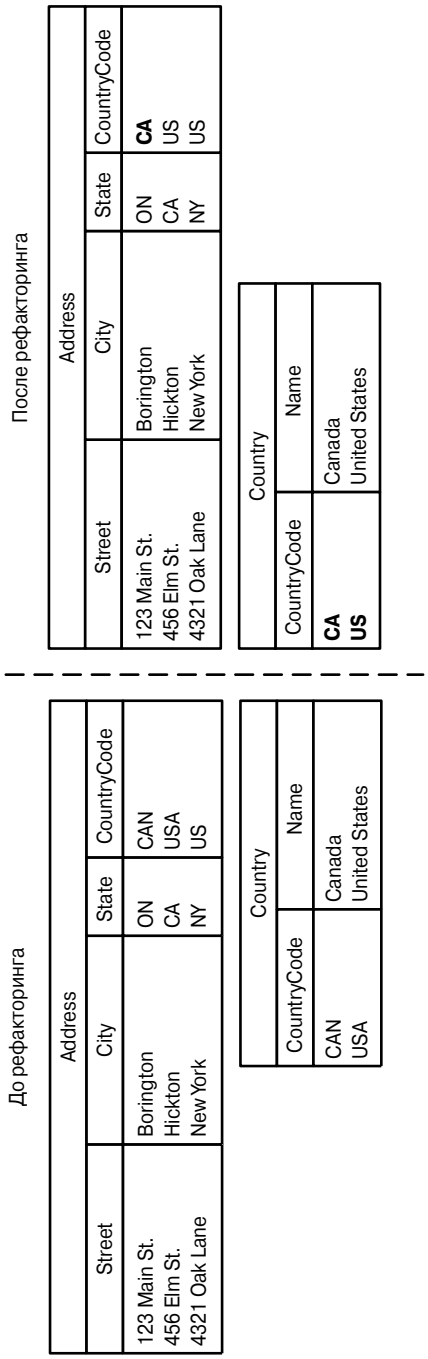


Рис. 7.2. Применение стандартных кодов штатов в таблице `StateFlower`

- **Уменьшение сложности кода.** Если для обозначения данных, которые являются семантически одинаковыми, применяется несколько разных значений, то приходится использовать лишний программный код, чтобы иметь возможность обрабатывать все эти различные значения. Например, существующий программный код `countryCode = 'US' || countryCode = 'USA' ...` можно было бы упростить примерно так: `countryCode = 'USA'`.

Потенциальные преимущества и недостатки

Задача стандартизации кодовых обозначений может оказаться сложной, поскольку такие обозначения часто используются во многих местах. Например, в некоторых таблицах какие-то кодовые обозначения могут использоваться в качестве внешнего ключа к другой таблице, поэтому необходимо стандартизировать не только источник данных, но и столбцы внешнего ключа. Кроме того, может оказаться, что кодовые обозначения реализованы в одном или нескольких приложениях в виде программных конструкций, в связи с чем потребуется проведение крупномасштабных обновлений. Например, предположим, что в приложениях, получающих доступ к таблице `Country`, в операторы SQL введены программные конструкции, содержащие значение `USA`, а в приложениях, использующих таблицу `Address`, содержатся программные конструкции с обозначением `US`.

Процедура обновления схемы

Чтобы применить операцию “Применение стандартных кодовых обозначений” к схеме базы данных, необходимо выполнить следующие действия.

1. **Определить применяемые стандартные обозначения.** Необходимо прежде всего определить “официально признанные” значения для всех кодовых обозначений. При этом необходимо узнать, поступают ли эти значения из существующих таблиц приложения или предоставляются пользователями бизнес-приложений. В любом случае рассматриваемые кодовые обозначения должны быть приняты другим лицом (лицами), участвующим в текущем проекте.
2. **Составить перечень таблиц, в которых хранятся кодовые обозначения.** Необходимо выявить все таблицы, которые включают столбец с кодовыми обозначениями. Для этого может потребоваться провести обширный анализ и выполнить много итераций, прежде чем удастся обнаружить все таблицы, в которых используются рассматриваемые кодовые обозначения. Следует отметить, что в определенный момент времени эта операция рефакторинга должна применяться только к одному столбцу, поэтому потенциально может потребоваться применить ее несколько раз для достижения согласованности данных во всей базе данных.
3. **Обновить хранимые процедуры.** После стандартизации кодовых обозначений может потребоваться обновить хранимые процедуры, которые получают доступ к столбцам, затронутым обновлением. Например, если в хранимой процедуре `getUSCustomerAddress` имеется конструкция `WHERE` с критерием `Address.CountryCode="US"`, то это выражение необходимо будет заменить выражением `Address.CountryCode="USA"`.

Процедура переноса данных

Если в процессе стандартизации вы остановитесь на каком-то конкретном кодовом обозначении, то нужно будет обновить все строки, в которых имеются нестандартные кодовые обозначения, чтобы перейти к стандартному обозначению. Если обновлению подлежит небольшое количество строк, то достаточно применить простой сценарий SQL, который обновит целевую таблицу (таблицы). Если же необходимо обновить большое количество данных или имеет место ситуация, в которой изменяется кодовое обозначение в транзакционных таблицах, то вместо этого следует использовать операцию “Обновление данных” (с. 329).

В следующем коде приведены операторы DML, позволяющие обновить данные в таблицах `Address` и `Country` в целях применения стандартных кодовых обозначений:

```
UPDATE Address SET CountryCode = 'CA' WHERE CountryCode = 'CAN';
UPDATE Address SET CountryCode = 'US' WHERE CountryCode = 'USA';

UPDATE Country SET CountryCode = 'CA' WHERE CountryCode = 'CAN';
UPDATE Country SET CountryCode = 'US' WHERE CountryCode = 'USA';
```

Процедура обновления программ доступа

В случае применения операции “Применение стандартных кодовых обозначений” необходимо проанализировать описанные ниже особенности внешних программ.

1. **Наличие предложений `WHERE`, в которых кодовые обозначения представлены в виде программных конструкций.** Может оказаться так, что требуют обновления операторы SQL, чтобы они содержали в своих предложениях `WHERE` правильные обозначения. Например, если в строке столбца `Country.Code` обозначение изменяется с `'US'` на `'USA'`, то необходимо откорректировать предложение `WHERE`, чтобы в нем использовалось новое обозначение.
2. **Применение кода проверки.** Аналогичным образом, может потребовать обновления исходный код, в котором проверяется достоверность значений атрибутов данных. Например, если в программе используется выражение `countryCode = 'US'`, то оно должно быть откорректировано в целях использования нового кодового обозначения.
3. **Применение поисковых конструкций.** Кодовые обозначения могут быть определены в различных программных “поисковых конструкциях”, таких как константы, перечисления или коллекции, чтобы их можно было использовать во всем приложении. Должны быть обновлены определения этих поисковых конструкций, что позволит применять новые кодовые обозначения.
4. **Применение тестового кода.** Кодовые обозначения часто бывают оформлены в виде программных конструкций в тестовых программных средствах и (или) других программных средствах выработки тестовых данных; необходимо откорректировать все эти программные конструкции, чтобы после этого в них использовались новые обозначения.

В следующем коде показано, как выглядит метод, применяемый для чтения почтовых адресов в США, до и после рефакторинга:

```
// Код до рефакторинга
stmt = DB.prepare("SELECT addressId, line1, line2, city, state,
zipcode, country FROM address WHERE countrycode = ?");
stmt.setString(1,"USA");
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt = DB.prepare("SELECT addressId, line1, line2, city, state,
zipcode, country FROM address WHERE countrycode = ?");
stmt.setString(1,"US");
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга "Применение стандартного типа"

Эта операция позволяет обеспечить согласованность типа данных столбца с типом данных других аналогичных столбцов в базе данных (рис. 7.3).

Обоснование

Операция рефакторинга "Применение стандартного типа" может применяться для выполнения описанных ниже действий.

- **Обеспечение ссылочной целостности.** Чтобы иметь возможность применить операцию "Добавление ограничения внешнего ключа" (с. 229) ко всем таблицам, хранящим информацию с одинаковым семантическим значением, необходимо предварительно стандартизировать типы данных отдельных столбцов. Например, на рис. 7.3 показано, как провести рефакторинг всех столбцов с номерами телефонов, чтобы данные в них хранились в виде целых чисел. Еще один распространенный пример обнаруживается, если, предположим, данные в столбце `Address.ZipCode` представлены с помощью типа данных `VARCHAR`, а в столбце `Customer.Zip` хранятся данные типа `NUMERIC`; необходимо провести стандартизацию этих данных с приведением к одному типу данных, чтобы иметь возможность применять ограничения ссылочной целостности.
- **Введение поисковой таблицы.** Если возникает необходимость в использовании операции "Добавление поисковой таблицы" (с. 185), то прежде всего требуется применить согласованный тип данных в рассматриваемых при этом двух столбцах с кодовыми обозначениями.
- **Обеспечение соответствия корпоративным стандартам.** Во многих организациях приняты детализированные стандарты данных и моделирования данных, которые обязаны соблюдать группы разработчиков. Часто при выполнении операции "Использование официально заданного источника данных" (с. 292) обнаруживается, что существующая в настоящее время схема данных не соответствует стандартам организации и поэтому должна быть подвергнута рефакторингу, чтобы в ней были отражены официально принятые кодовые обозначения для источника данных.

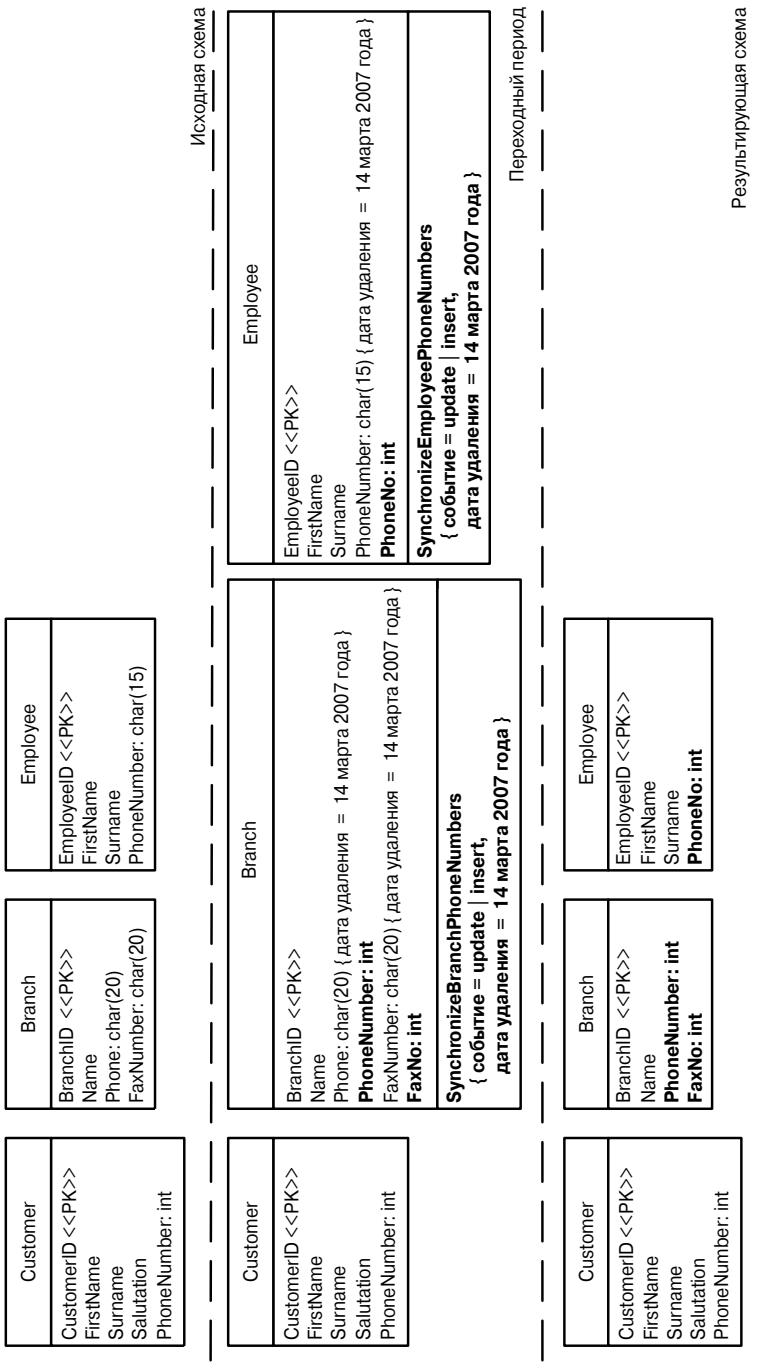


Рис. 7.3. Применение стандартных типов данных в таблицах `Customer`, `Branch` *и* `Employee`

- **Уменьшение сложности кода.** Если для обозначения данных, которые семантически являются одинаковыми, применяется несколько разных значений, то приходится использовать лишний программный код, чтобы иметь возможность обрабатывать все эти различные типы столбцов. Например, рефакторингу может быть подвергнут код проверки номеров телефонов для классов `Customer`, `Branch` и `Employee` в целях внедрения одного совместно используемого метода.

Потенциальные преимущества и недостатки

Задача стандартизации типов данных может оказаться сложной, поскольку ссылки на отдельные столбцы часто встречаются во многих местах. Например, если на рассматриваемый столбец ссылается несколько классов приложений, то после изменения типа данных столбца может потребоваться корректировка кода этих классов. Кроме того, при попытке изменить тип данных столбца может возникнуть такая ситуация, что данные источника не удастся привести к типу данных назначения. Например, при преобразовании типа данных столбца `Customer.Zip` к типу данных `NUMERIC` будет исключена возможность преобразовывать международные почтовые коды, такие как `R2D 2C3`, которые содержат символьные данные.

Процедура обновления схемы

Чтобы иметь возможность применить эту операцию рефакторинга, вначале необходимо определить стандартный тип данных. Для этого требуется принять решение о том, каким должен быть "официально признанный" тип данных для рассматриваемых столбцов. Этот тип данных должен обеспечивать обработку всех существующих данных, а внешние программы доступа должны обрабатывать данные этого типа. (Иногда более старые языки программирования не обеспечивают обработку данных нового типа.) Затем необходимо определить перечень таблиц, которые включают столбец (столбцы), содержащий тип данных, предназначенный для изменения. Для этого может потребоваться провести обширный анализ и выполнить много итераций, прежде чем удастся обнаружить все таблицы, в которых имеются столбцы, подлежащие изменению. Следует отметить, что в определенный момент времени эта операция рефакторинга должна применяться только к одному столбцу, поэтому потенциально может потребоваться применить ее несколько раз для обеспечения согласованности данных во всей базе данных.

На рис. 7.3 показано, как внести изменения в столбцы `Branch.Phone`, `Branch.FaxNumber` и `Employee.PhoneNumber`, чтобы применить один и тот же целочисленный тип данных. Поскольку столбец `Customer.PhoneNumber` уже имеет целочисленный тип данных, он не должен подвергаться рефакторингу. Безусловно, эти три операции рефакторинга фактически являются отдельными, но все три рассматриваемых столбца относятся к одной и той же таблице, поэтому авторы сочли необходимым увязать друг с другом все эти операции рефакторинга.

В следующем примере кода показаны три операции рефакторинга, необходимые для внесения изменений в столбцы `Branch.Phone`, `Branch.FaxNumber` и `Employee.Phone`. Кроме того, в рассматриваемые таблицы решено ввести новый столбец с использованием операции "Введение нового столбца" (с. 321). Следует отметить, что необходимо предусмотреть определенное время для того, чтобы все приложения можно было перевести на

использование новых столбцов, поэтому на протяжении переходного этапа решено сопровождать столбцы обоих типов, а также синхронизировать в них данные.

```
ALTER TABLE Branch ADD COLUMN PhoneNumber INT;
COMMENT ON Branch.PhoneNumber "Replaces Phone, dropdate=2007-03-27"

ALTER TABLE Branch ADD COLUMN FaxNo INT;
COMMENT ON Branch.FaxNo "Replaces FaxNumber, dropdate=2007-03-27"

ALTER TABLE Employee ADD PhoneNo INT;
COMMENT ON Employee.PhoneNo "Replaces PhoneNumber, dropdate=2007-03-27"
```

В следующем коде показано, как синхронизировать изменения в столбцах `Branch.Phone`, `Branch.FaxNumber` и `Employee.Phone` с существующими столбцами:

```
CREATE OR REPLACE TRIGGER SynchronizeBranchPhoneNumbers
  BEFORE INSERT OR UPDATE
  ON Branch
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF :NEW.PhoneNumber IS NULL THEN
      :NEW.PhoneNumber := :NEW.Phone;
    END IF;
    IF :NEW.Phone IS NULL THEN
      :NEW.Phone := :NEW.PhoneNumber;
    END IF;
    IF :NEW.FaxNumber IS NULL THEN
      :NEW.FaxNumber := :NEW.FaxNo;
    END IF;
    IF :NEW.FaxNo IS NULL THEN
      :NEW.FaxNo := :NEW.FaxNumber;
    END IF;
  END;
/

CREATE OR REPLACE TRIGGER SynchronizeEmployeePhoneNumbers
  BEFORE INSERT OR UPDATE
  ON Employee
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF :NEW.PhoneNumber IS NULL THEN
      :NEW.PhoneNumber := :NEW.PhoneNo;
    END IF;
    IF :NEW.PhoneNo IS NULL THEN
      :NEW.PhoneNo := :NEW.PhoneNumber;
    END IF;
  END;
/

-- Первоначальное обновление существующих данных
```

```
UPDATE Branch SET
PhoneNumber = formatPhone(Phone),
FaxNo = formatPhone(FaxNumber);
UPDATE Employee SET
  PhoneNo = formatPhone(PhoneNumber);

-- Удаление старых столбцов 23 марта 2007 года
ALTER TABLE Branch DROP COLUMN Phone;
ALTER TABLE Branch DROP COLUMN FaxNumber;
ALTER TABLE Employee DROP COLUMN PhoneNumber;
DROP TRIGGER SynchronizeBranchPhoneNumbers;
DROP TRIGGER SynchronizeEmployeePhoneNumbers;
```

Процедура переноса данных

Если преобразования требует небольшое количество строк, то может оказаться, что достаточно воспользоваться простым сценарием SQL для преобразования данных целевого столбца (столбцов). Если же требуется преобразовать большие количества данных или сами преобразования данных являются сложными, то целесообразно применить операцию "Обновление данных" (с. 329).

Процедура обновления программ доступа

При использовании операции "Применение стандартного типа" необходимо обновить внешние программы, как описано ниже.

1. **Перейти к использованию переменных приложения с другими типами данных.** Необходимо откорректировать код программы так, чтобы применяемые в нем типы данных соответствовали типу данных столбца.
2. **Откорректировать код взаимодействия с базой данных.** Необходимо обновить код, обеспечивающий сохранение, удаление и выборку данных из рассматриваемого столбца, для работы с новым типом данных. Например, если изменился тип данных столбца `Customer.Zip` в результате перехода от символьного к числовому типу данных, то необходимо откорректировать код приложения, заменив выражение `customerGateway.getString("ZIP")` выражением `customerGateway.getLong("ZIP")`.
3. **Откорректировать код бизнес-логики.** Аналогичным образом, может потребоваться обновить прикладной код, который работает с рассматриваемым столбцом. Например, код оператора сравнения, такой как `Branch.Phone = 'XXX-XXXX'`, должен быть обновлен, чтобы он выглядел примерно как `Branch.Phone = XXXXXXX`.

В следующем фрагменте показан код класса до и после рефакторинга, в котором осуществляется поиск в строке `Branch` заданного значения `BranchID`; код откорректирован в связи с переходом типа данных столбца `PhoneNumber` от `Long` к `String`:

```
// Код до рефакторинга
stmt = DB.prepare("SELECT BranchId, Name, PhoneNumber, "
            "FaxNumber FROM branch WHERE BranchId = ?");
```

```
stmt.setLong(1,findBranchId);
stmt.execute();
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
  rs.getLong("BranchId");
  rs.getString("Name");
  rs.getString("PhoneNumber");
  rs.getString("FaxNumber");
}

// Код после рефакторинга
stmt = DB.prepare("SELECT BranchId, Name, PhoneNumber, "+
            "FaxNumber FROM branch WHERE branchId = ?");
stmt.setLong(1,findBranchId);
stmt.execute();
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
  rs.getLong("BranchId");
  rs.getString("Name");
  rs.getLong("PhoneNumber");
  rs.getString("FaxNumber");
}
```

Операция рефакторинга “Осуществление стратегии консолидированных ключей”

Эта операция позволяет выбрать единую стратегию определения ключей для всех сущностей и предусмотреть ее последовательное применение во всей базе данных (рис. 7.4).

Обоснование

Характерная особенность многих сущностей, применяемых в деловых приложениях, состоит в том, что в них имеется несколько потенциальных ключей. Обычно для делового предприятия смысл имеет один или несколько ключей, кроме того, всегда можно ввести в таблицу столбец суррогатного ключа. Например, предположим, что в таблице `Customer` применяется столбец `CustomerPOID` в качестве первичного ключа, а также столбцы `CustomerNumber` и `SocialSecurityNumber` в качестве альтернативных и (или) вторичных ключей. Как описано ниже, необходимость в применении операции “Осуществление стратегии консолидированных ключей” может возникнуть по нескольким причинам.

- **Повышение производительности.** Возможно, с каждым ключом связан индекс, для которого необходимо обеспечить эффективное выполнение операций вставки, обновления и удаления в базе данных.
- **Обеспечение соответствия корпоративным стандартам.** Во многих организациях приняты подробные рекомендации по определению и моделированию данных, которых должны придерживаться группы разработчиков, а также рекомендации, в которых указаны предпочтительные ключи для сущностей. Часто при использовании операции “Использование официально заданного источника данных” (с. 292) обнаруживается, что текущая схема данных не соответствует принятым стандартам и поэтому

должна быть подвергнута рефакторингу с тем, чтобы в ней были отражены кодовые обозначения, соответствующие официально принятому источнику данных.

- **Улучшение согласованности кода.** Если для работы с одной и той же сущностью применяются разнообразные ключи, то фактически эксплуатируется код, в котором доступ к таблице осуществляется многими способами. В результате возрастает нагрузка по сопровождению для тех пользователей, кто работает с этим кодом, поскольку им приходится вникать в нюансы каждого используемого подхода.

Потенциальные преимущества и недостатки

При осуществлении стратегий консолидации ключей могут возникать затруднения. Например, как показано на рис. 7.4, необходимо не только обновить схему таблицы `Policy`, но и схемы всех таблиц, которые включают внешние ключи, ссылающиеся на таблицу `Policy`, и характеризуются тем, что в них не используется выбранная стратегия определения ключей. Для этого требуется применить операцию "Замена столбца" (с. 160). Может быть также обнаружено, что существующий набор ключей не содержит такого ключа, который в полной степени отвечал бы требованию иметь "один, истинный ключ", и поэтому необходимо применить либо операцию "Введение суррогатного ключа" (с. 123), либо "Введение индекса" (с. 271).

Процедура обновления схемы

Для реализации рассматриваемой операции рефакторинга в существующей схеме базы данных необходимо выполнить следующие действия.

1. **Выявить подходящий ключ.** Необходимо остановиться на одном из решений, касающихся того, какой ключевой столбец (столбцы) для рассматриваемой сущности должен считаться "официально признанным". В идеальном случае это решение должно соответствовать корпоративным стандартам данных, если таковые имеются.
2. **Обновить схему исходной таблицы.** Самый простой подход состоит в том, чтобы использовать текущий первичный ключ и прекратить использование альтернативных ключей. Осуществляя эту стратегию, достаточно просто удалить индексы, поддерживающие ключ, если таковые имеются. Этот подход останется осуществимым, даже если будет решено применять один из альтернативных ключей, а не текущий первичный ключ. Но если ни один из существующих ключей не применим, то может потребоваться осуществить операцию "Введение суррогатного ключа" (с. 123).
3. **Обозначить ненужные ключи как устаревшие.** Для всех существующих ключей, не являющихся первичными (в рассматриваемом случае — `PolicyNumber`), должно быть отмечено, что они больше не будут использоваться как ключи после завершения переходного периода. Следует отметить, что может оказаться целесообразным оставить в силе ограничения уникальности на этих столбцах, даже если больше не предусмотрено применение рассматриваемых столбцов в качестве ключей.

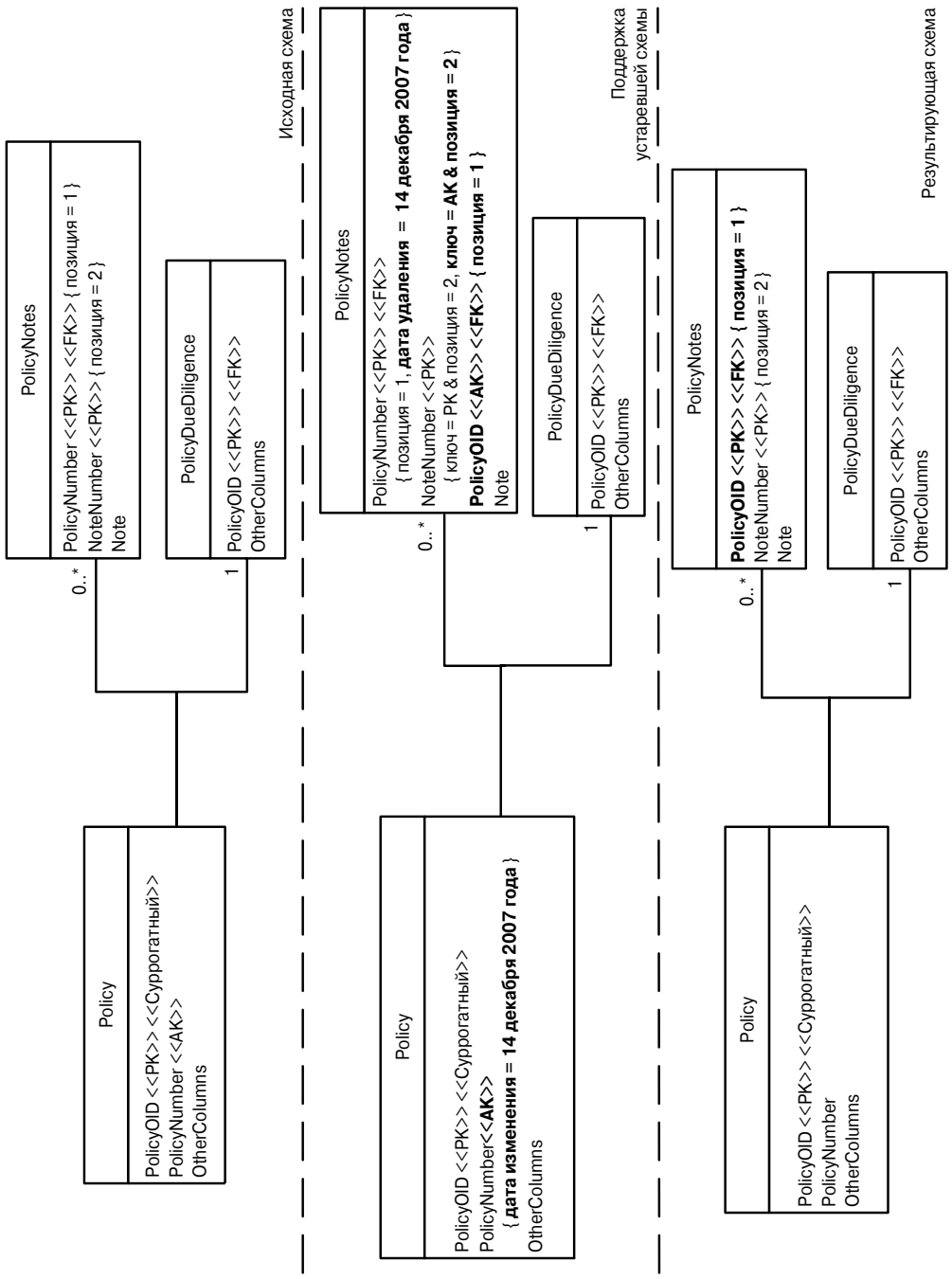


Рис. 7.4. Осуществление стратегии консолидации ключей применительно к таблице `Policy`

4. **Добавить новый индекс.** Если таковой еще не существует, то (в рассматриваемом примере) с помощью операции "Введение индекса" (с. 271) для таблицы `Policy` должен быть введен новый индекс в соответствии с потребностями в использовании официально принятого ключа.

На рис. 7.4 показано, как можно консолидировать стратегию использования ключей для таблицы `Policy` с тем, чтобы в этой таблице в качестве ключа эксплуатировался только столбец `PolicyOID`. Для этого необходимо обозначить столбец `Policy.PolicyNumber` как устаревший для указания на то, что этот столбец больше не будет использоваться в качестве ключа начиная с 14 декабря 2007 года, а также ввести столбец `PolicyNotes.PolicyOID` как новый ключевой столбец для замены `PolicyNotes.PolicyNumber`.

```
ALTER TABLE PolicyNotes ADD PolicyOID CHAR(16);
```

Следующий код должен быть выполнен по завершении переходного периода для удаления столбца `PolicyNotes.PolicyNumber` и индекса для альтернативного ключа, основанного на столбце `Policy.PolicyNumber`:

```
COMMENT ON Policy 'Consolidation of keys to use only PolicyPOID,
dropdate = <<2007-12-14>>'
DROP INDEX PolicyIndex2;

COMMENT ON PolicyNotes 'Consolidation of keys to use only PolicyPOID,
therefore drop the PolicyNumber column, dropdate = <<2007-12-14>>'

ALTER TABLE PolicyNotes ADD CONSTRAINT
  PolicyNotesPolicyOID_PK
  PRIMARY KEY (PolicyOID, NoteNumber);

ALTER TABLE PolicyNotes DROP COLUMN PolicyNumber;
```

Процедура переноса данных

После проведения указанной операции рефакторинга в таблице с внешними ключами, поддерживающими связи с таблицей `Policy`, должны быть реализованы внешние ключи, которые отражают выбранную стратегию использования ключей. Например, в таблице `PolicyNotes` первоначально был реализован внешний ключ, основанный на столбце `Policy.PolicyNumber`. А теперь должен быть реализован внешний ключ, основанный на столбце `Policy.PolicyOID`. Из этого следует, что для осуществления указанного преобразования может потребоваться применение операции "Замена столбца" (с. 160) и что для рассматриваемой операции рефакторинга нужно будет скопировать данные из исходного столбца (значение в столбце `Policy.PolicyOID`) в столбец `PolicyNotes.PolicyOID`. Значения для столбца `PolicyNotes.PolicyNumber` задаются в следующем коде:

```
UPDATE PolicyNotes
            SET PolicyNotes.PolicyOID = Policy.PolicyOID
  WHERE PolicyNotes.PolicyNumber = Policy.PolicyNumber;
```

Процедура обновления программ доступа

В ходе применения рассматриваемой операции рефакторинга необходимо прежде всего обеспечить, чтобы в конструкциях `WHERE` существующих операторов SQL были указаны официально утвержденные столбцы первичного ключа в целях обеспечения и в дальнейшем высокой производительности операций соединения. Например, допустим, что в коде, применявшемся до рефакторинга, соединение таблиц `Policy`, `PolicyNotes` и `PolicyDueDiligence` осуществлялось с использованием комбинации столбцов `PolicyOID` и `PolicyNumber`. А в коде, который вводится в действие после осуществления операции рефакторинга, для соединения таблиц служит исключительно столбец `PolicyOID`:

```
// Код до рефакторинга
stmt.prepare(
 "SELECT Policy.Note FROM Policy, PolicyNotes " +
 "WHERE Policy.PolicyNumber = PolicyNotes.PolicyNumber "+
 "AND Policy.PolicyOID=?");
stmt.setLong(1,policyOIDToFind);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
 "SELECT Policy.Note FROM Policy, PolicyNotes " +
 "WHERE Policy.PolicyOID = PolicyNotes.PolicyOID "+
 "AND Policy.PolicyOID=?");
stmt.setLong(1,policyOIDToFind);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга “Уничтожение ограничения столбца”

Эта операция позволяет удалить ограничение столбца из существующей таблицы (рис. 7.5).

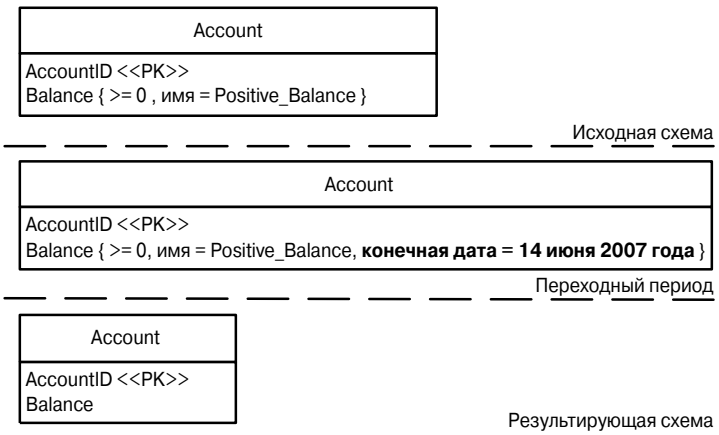


Рис. 7.5. Удаление ограничения столбца из таблицы `Account`

Если требуется удалить ограничение ссылочной целостности, относящееся к одному из столбцов, то следует предусмотреть возможность использования операции “Уничтожение ограничения внешнего ключа” (с. 238).

Обоснование

Наиболее распространенной причиной, по которой может потребоваться применить операцию “Уничтожение ограничения столбца”, является то, что рассматриваемое ограничение становится больше не применимым из-за изменений в бизнес-правилах. Например, допустим, что на столбец `Address.Country` в настоящее время распространяется ограничение, согласно которому в нем могут присутствовать лишь значения `"US"` и `"Canada"`, но теперь компания перешла к ведению деловых отношений во всем мире, поэтому не исключена возможность, что ее адресаты будут находиться в разных странах мира. Еще одна причина может быть обусловлена тем, что рассматриваемое ограничение применимо только к подмножеству приложений, имеющих доступ к конкретному столбцу, возможно, из-за того, что в одни приложения были внесены определенные изменения, а в другие — нет. Например, некоторые приложения могут использоваться в международных деловых операциях, а другие все еще ограничиваются Северной Америкой; поскольку ограничение больше не применимо ко всем приложениям, его следует удалить из общей базы данных и реализовать в приложениях, в которых оно действительно требуется. В качестве побочного эффекта удаления ограничения может обнаружиться некоторое незначительное повышение производительности, связанное с тем, что работа по принудительной поддержке ограничения в базе данных больше не выполняется. Еще одна причина состоит в том, что столбец был подвергнут рефакторингу, т.е. возможно, что к нему была применена операция “Применение стандартных кодовых обозначений” (с. 188) или “Применение стандартного типа” (с. 192), и теперь необходимо удалить ограничение столбца, подвергнуть это ограничение рефакторингу, а затем снова ввести в действие с помощью операции “Введение ограничения столбца” (с. 207).

Потенциальные преимущества и недостатки

Основные затруднения, связанные с выполнением этой операции рефакторинга, обусловлены тем, что в принципе может потребоваться реализовать логику ограничения в подмножестве приложений, для которых необходима поддержка этого ограничения. А поскольку одно и то же программное средство реализуется в нескольких местах, то возникает риск, что такая реализация будет выполнена в разных местах немного по-разному.

Процедура обновления схемы

В рассматриваемом примере для обновления схемы базы данных необходимо с помощью конструкции `DROP CONSTRAINT` команды `ALTER TABLE` языка SQL удалить ограничение, заданное на столбце `Balance`. На рис. 7.5 показан пример, в котором ограничение состоит в том, что значение в столбце `Account.Balance` должно быть положительным; это ограничение удаляется, чтобы иметь возможность применять к счетам операции овердрафта. В следующем коде приведены операторы DDL, предназначенные для удаления ограничения, заданного на столбце `Account.Balance`:

```
ALTER TABLE Account DROP CONSTRAINT Positive_Balance;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных не требуется переносить какие-либо данные.

Процедура обновления программ доступа

Программы доступа, работающие с этим столбцом, должны включать программные средства обработки любых ошибок, активизируемых базой данных, если данные, записываемые в столбец, не соответствуют ограничению. Этот код необходимо удалить из программ.

Операция рефакторинга “Уничтожение значения, заданного по умолчанию”

Эта операция позволяет удалить из существующего столбца таблицы заданное по умолчанию значение, которое предусмотрено в базе данных (рис. 7.6).

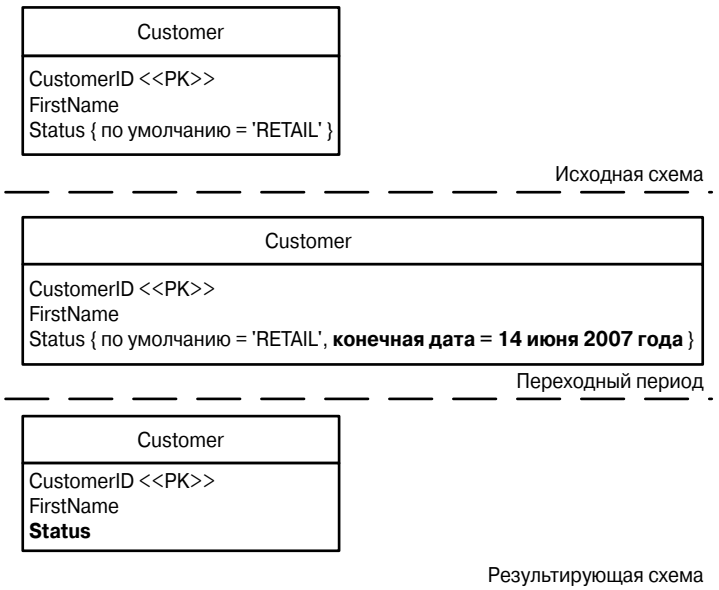


Рис. 7.6. Удаление заданного по умолчанию значения для столбца `Customer.Status`

Обоснование

Операция “Введение заданного по умолчанию значения” часто используется (с. 213), если необходимо, чтобы база данных неизменно заполняла некоторые столбцы данными, притом что в эти столбцы не передаются данные из приложения. Но во многих случаях необходимость в том, чтобы базой данных по-прежнему выполнялась вставка данных в некоторые из столбцов, отпадает, поскольку все необходимые данные поступают из при-

ложения, поэтому приходится отменять вставку данных, предусмотренных по умолчанию. В такой ситуации используется операция рефакторинга “Уничтожение значения, заданного по умолчанию”.

Потенциальные преимущества и недостатки

При выполнении этой операции рефакторинга необходимо учитывать два потенциальных затруднения. Во-первых, могут возникнуть непредвиденные побочные эффекты. При создании некоторых приложений могут быть приняты предположения, что база данных неизменно вставляет заданное по умолчанию значение, в связи с этим такие приложения могут действовать иначе после обнаружения того, что столбцы во вновь вставленных строках, которые прежде содержали бы какое-то значение, теперь содержат NULL-значение. Во-вторых, возникает необходимость улучшить обработку исключительных ситуаций во внешних программах, что может потребовать неоправданно больших усилий. Если столбец не допускает наличия NULL-значений, а данные не предоставляются приложением, то база данных активизирует исключение, на обработку которого приложение не рассчитано.

Процедура обновления схемы

Для обновления схемы в целях удаления применяемого по умолчанию значения необходимо удалить заданное по умолчанию значение из определения столбца таблицы в базе данных с использованием конструкции `MODIFY` команды `ALTER TABLE`. В приведенном ниже коде показаны этапы удаления заданного по умолчанию значения из столбца `Customer.Status`, показанного на рис. 7.6. С точки зрения эксплуатации базы данных наличие заданного по умолчанию NULL-значения равносильно отсутствию заданного по умолчанию значения.

```
ALTER TABLE Customer MODIFY Status DEFAULT NULL;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных не требуется переносить какие-либо данные.

Процедура обновления программ доступа

Если от применения заданных по умолчанию значений в рассматриваемой таблице зависит работа некоторых программ доступа, то необходимо либо ввести код проверки данных с учетом указанного изменения в таблице, либо рассмотреть возможность отказаться от осуществления этой операции рефакторинга.

В следующем примере показано, как можно предусмотреть в прикладном коде предоставление значения для столбца после выполнения рассматриваемой операции рефакторинга, чтобы была исключена зависимость от базы данных, предоставлявшей ранее значения, применяемые по умолчанию:

```
// Код до рефакторинга
public void createRetailCustomer
  (long customerId,String firstName) {
```

```
  stmt = DB.prepare("INSERT into customer" +
    "(Customerid, FirstName) " +
    "values (?, ?)");
  stmt.setLong(1, customerId);
  stmt.setString(2, firstName);
  stmt.execute();
}

// Код после рефакторинга
public void createRetailCustomer
  (long customerId,String firstName) {
  stmt = DB.prepare("INSERT into customer" +
    "(Customerid, FirstName, Status) " +
    "values (?, ?, ?)");
  stmt.setLong(1, customerId);
  stmt.setString(2, firstName);
  stmt.setString(3, RETAIL);
  stmt.execute();
}
```

Операция рефакторинга “Уничтожение столбца, не допускающего NULL-значений”

Эта операция позволяет модифицировать существующий столбец, не допускающий NULL-значений, чтобы он мог принимать NULL-значения (рис. 7.7).

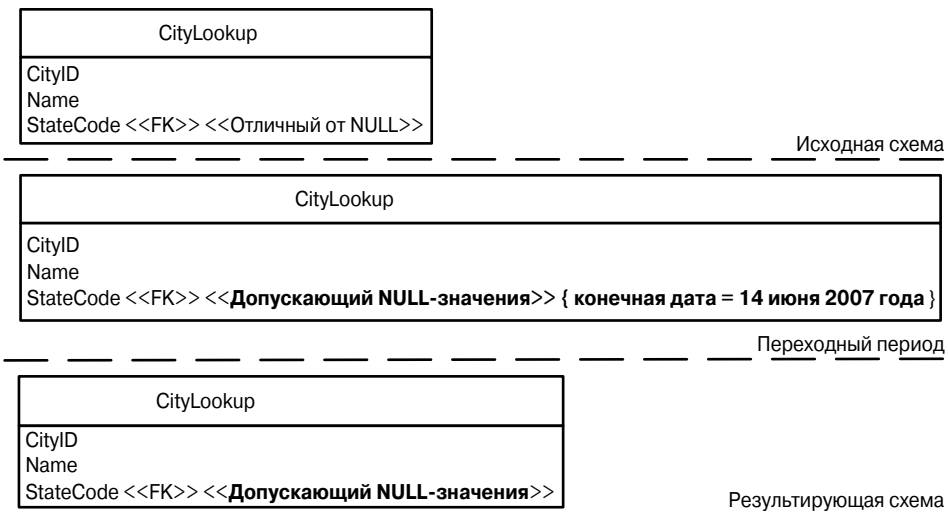


Рис. 7.7. Преобразование столбца `CityLookup.StateID` в столбец, принимающий NULL-значения

Обоснование

Необходимость в применении операции "Уничтожение столбца, не допускающего NULL-значений" может быть обусловлена двумя основными причинами. Во-первых, может произойти такое изменение бизнес-процесса, что ввод данных, соответствующих различным частям деловой сущности, должен осуществляться в разное время. Например, сущность может создаваться в одном приложении, но при этом значение одному из ее столбцов не будет присваиваться сразу же, а в другом приложении соответствующая строка обновляется в дальнейшем. Во-вторых, может возникнуть необходимость, чтобы в течение переходного периода какой-то конкретный столбец мог принимать NULL-значения. Например, если в одном из приложений невозможно предоставить значение для рассматриваемого столбца в связи с тем, что это приложение проходит через какую-то операцию рефакторинга, то может потребоваться отменить ограничение, касающееся запрета применения NULL-значений, на какое-то ограниченное время в течение переходного периода, чтобы приложение могло продолжать работать. А в дальнейшем применяется операция "Преобразование столбца в недопускающий NULL-значения" (с. 216) для повторного преобразования ограничения в ту форму, которую оно имело до этого.

Потенциальные преимущества и недостатки

В каждом приложении, получающем доступ к рассматриваемому столбцу, необходимо предусмотреть возможность принимать NULL-значения, пусть даже в этом приложении NULL-значения будут просто игнорироваться или, что более вероятно, будет предусматриваться использование более подходящего заданного по умолчанию значения после обнаружения того, что в столбце содержится NULL-значение. Если должно применяться какое-то подходящее значение, заданное по умолчанию, то целесообразно также рассмотреть возможность использования операции рефакторинга "Введение заданного по умолчанию значения" (с. 213).

Процедура обновления схемы

Для выполнения операции "Уничтожение столбца, не допускающего NULL-значений" достаточно удалить ограничение `NOT NULL`, распространявшееся на рассматриваемый столбец. Такое действие выполняется с помощью команды `ALTER TABLE MODIFY COLUMN` языка SQL. Ниже приведен код, который показывает, как вводить NULL-значения в столбец `CityLookup.StateCode`.

```
-- 14 июня 2007 года
ALTER TABLE CityLookup MODIFY StateCode NULL;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных не требуется переносить какие-либо данные.

Процедура обновления программ доступа

Необходимо подвергнуть рефакторингу все внешние программы, которые в настоящее время получают доступ к столбцу `CityLookup.StateCode`, чтобы в коде этих программ могла должным образом осуществляться обработка NULL-значений. После проведения рассматриваемой операции рефакторинга появляется возможность вводить в базу данных NULL-значения, поэтому может осуществляться и выборка таких значений, а из этого следует, что должен быть дополнительно введен код проверки на наличие NULL-значений. Кроме того, необходимо переопределить весь код, в котором производится проверка на наличие исключительных ситуаций, вызванных обнаружением NULL-значений, поскольку такие исключения больше не активизируются базой данных.

В следующем примере кода показано, как ввести программные средства проверки на наличие NULL-значений в прикладной код:

```
// Код до рефакторинга
public StringBuffer getAddressString(Address address) {
  StringBuffer stringAddress = new StringBuffer();
  stringAddress = address.getStreetLine1();
  stringAddress.append(address.getStreetLine2());
  stringAddress.append(address.getCity());
  stringAddress.append(address.getPostalCode());
  stringAddress.append(states.getNameFor(address.getStateCode()));
  return stringAddress;
}

// Код после рефакторинга
public StringBuffer getAddressString(Address address) {
  StringBuffer stringAddress = new StringBuffer();
  stringAddress = address.getStreetLine1();
  stringAddress.append(address.getStreetLine2());
  stringAddress.append(address.getCity());
  stringAddress.append(address.getPostalCode());
  String statecode = address.getStateCode;
  if (statecode != null) {
    stringAddress.append(states.getNameFor(stateCode()));
  }
  return stringAddress;
}
```

Операция рефакторинга “Введение ограничения столбца”

Эта операция позволяет ввести ограничение столбца в существующую таблицу (рис. 7.8).

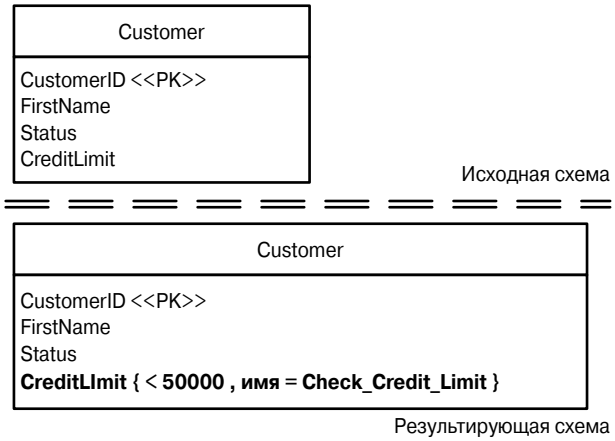


Рис. 7.8. Введение ограничения, распространяющегося на столбец `Customer.CreditLimit`

Если есть необходимость распространить на какой-то столбец ограничения, касающиеся запрета использования NULL-значений, то может быть предусмотрено применение операции рефакторинга "Преобразование столбца в недопускающий NULL-значения" (с. 216). Если же необходимо распространить на столбец ограничение ссылочной целостности, то целесообразно предусмотреть использование операции "Добавление ограничения внешнего ключа" (с. 229).

Обоснование

Чаще всего необходимость в применении операции "Введение ограничения столбца" может быть обусловлена тем, что требуется обеспечить ввод в столбец допустимых данных всеми приложениями, взаимодействующими с базой данных. Иными словами, ограничения столбца обычно используются для реализации общих и вместе с тем относительно простых правил проверки данных.

Потенциальные преимущества и недостатки

Основная проблема состоит в определении того, действительно ли рассматриваемое ограничение является общим для этого элемента данных применительно ко всем программам, которые получают к нему доступ. В отдельных приложениях могут быть предусмотрены собственные, уникальные версии ограничения для такого столбца. Например, предположим, что в таблице `Customer` имеется столбец `FavoriteColor`. Корпоративный стандарт может предусматривать использование в этом столбце значений `"Red"`, `"Green"` и `"Blue"`, но по каким-то весомым причинам в одном из приложений может быть предусмотрена возможность применения в качестве обозначения цвета четвертого значения — `"Yellow"`, а еще в двух приложениях допускается использовать только `"Red"` и `"Blue"`. Такой факт можно оспаривать, утверждая, что во всех приложениях должен применяться один согласованный стандарт, но реальность такова, что по многим важным причинам в отдельных приложениях допускается отклонение от этого стандарта. Из этого следует, что в отдельных приложениях могут быть реализованы собственные

версии бизнес-правил, поэтому даже в самых лучших обстоятельствах будет обнаруживаться, что в базе данных имеются данные, не соответствующие условиям ограничения. Один из подходов к устранению этого затруднения может состоять в применении сценария, в котором последовательно обрабатываются все таблицы базы данных, а затем подготавливаются отчеты обо всех нарушениях ограничений, с указанием данных, которые должны быть исправлены. Такие процессы могут эксплуатироваться в пакетном режиме в те часы, когда отсутствует пиковая нагрузка на базу данных и приложения.

Рассматриваемая операция рефакторинга может также применяться в связи с необходимостью подвергнуть рефакторингу существующее ограничение столбца. Такие операции рефакторинга, как “Применение стандартных кодовых обозначений” (с. 188) и “Применение стандартного типа” (с. 192), могут потребовать, соответственно, откорректировать основополагающее кодовое обозначение или тип данных для столбца, что повлечет за собой необходимость повторно ввести ограничение.

Процедура обновления схемы

Как показано на рис. 7.8, чтобы обновить схему базы данных, необходимо ввести ограничение на столбце, в данном случае `Customer.CreditLimit`. А в приведенном ниже коде можно видеть, что такое действие осуществляется с помощью конструкции `ADD CONSTRAINT` команды `ALTER TABLE` языка SQL. В данном случае лимит кредита должен быть меньше 50 тыс. долл., чтобы можно было предотвратить мошенничество.

```
ALTER TABLE Customer ADD CONSTRAINT
  Check_Credit_Limit CHECK (CreditLimit < 50000.00);
```

Процедура переноса данных

При использовании этой операции рефакторинга перенос данных как таковой не производится, но возникает необходимость решить другую сложную проблему — обеспечить соответствие существующих данных ограничению еще до того, как это ограничение будет применено к столбцу. Вначале необходимо определить само ограничение, проведя работу с другими лицами, заинтересованными в создании проекта, для выявления того, какие действия должны быть выполнены со значениями данных, которые не соответствуют применяемому ограничению. После этого следует исправить исходные данные. Может также потребоваться применить в случае необходимости операцию “Обновление данных” (с. 329), чтобы значения, хранящиеся в рассматриваемом столбце, соответствовали ограничению.

Процедура обновления программ доступа

Необходимо обеспечить, чтобы программы доступа могли обрабатывать любые ошибки, активизируемые базой данных в тех случаях, когда данные, записываемые в столбец, не соответствуют ограничению. Требуется выполнить поиск каждой точки в программе, в которой выполняется вставка или обновление в таблице, а затем ввести соответствующий код обработки исключительной ситуации, как показано в следующем примере:

```
// Код до рефакторинга
stmt = conn.prepare(
  "INSERT INTO Customer "+
```

```
  "(CustomerID,FirstName,Status,CreditLimit) "+
  "VALUES (?,?,?,?)");
stmt.setLong(1,customerId);
stmt.setString(2,firstName);
stmt.setString(3,status);
stmt.setBigDecimal(4,creditLimit);
stmt.executeUpdate();
}

// Код после рефакторинга
stmt = conn.prepare(
  "INSERT INTO Customer "+
  "(CustomerID,FirstName,Status,CreditLimit) "+
  "VALUES (?,?,?,?)");
stmt.setLong(1,customerId);
stmt.setString(2,firstName);
stmt.setString(3,status);
stmt.setBigDecimal(4,creditLimit);
try {
  stmt.executeUpdate();
} catch (SQLException exception){
    while (exception != null) {
      int errorCode = e.getErrorCode();
      if (errorCode = 2290) {
            handleCreditLimitExceeded();
      }
      }
}
```

Операция рефакторинга "Введение общего формата"

Эта операция применяется для приведения всех значений данных в существующем столбце таблицы к согласованному формату (рис. 7.9).

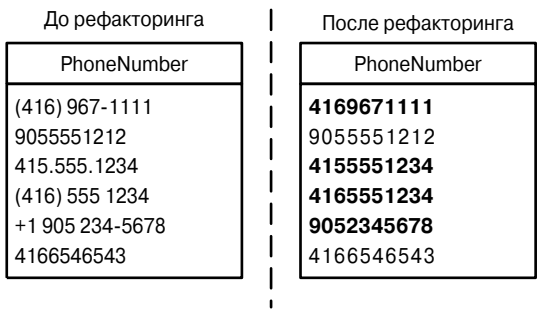


Рис. 7.9. Применение общего формата к столбцу `Address.PhoneNumber`

Обоснование

Операция “Введение общего формата” обычно используется для упрощения кода внешней программы. Если одни и те же данные хранятся в различных форматах, то во внешних программах для работы с такими данными приходится применять дополнительные сложные программные средства, без которых можно было бы обойтись. Обычно лучше предусмотреть хранение всех данных в унифицированном формате, для того чтобы в приложениях, взаимодействующих с базой данных, не приходилось учитывать наличие нескольких разных форматов. Например, если значения в столбце `Customer.Phonenumber` хранятся в виде `'4163458899'`, `'905-777-8889'` и `'(990) 345-6666'`, то в каждом приложении, получающем доступ к этому столбцу, должна быть предусмотрена возможность выполнять синтаксический анализ всех трех форматов. Эта проблема еще больше усугубляется, если во внешней программе приходится проводить синтаксический анализ данных в нескольких местах, чаще всего потому, что программа плохо спроектирована. Применение единого формата позволяет уменьшить объем кода проверки, относящегося к столбцу `Customer.Phonenumber`, а также перейти к использованию менее сложных программных средств вывода данных, поскольку все номера телефонов будут храниться, допустим, как `1234567890`, а передаваться в отчет как `(123) 456-7890`.

Необходимость в использовании этой операции рефакторинга часто обусловлена также стремлением привести данные в соответствие с существующими корпоративными стандартами. Часто при использовании операции “Использование официально заданного источника данных” (с. 292) обнаруживается, что текущий формат данных отличается от “официально принятого” для рассматриваемого источника данных, поэтому данные должны быть переформатированы для обеспечения согласованности со стандартом.

Потенциальные преимущества и недостатки

Если в одном и том же столбце данные представлены в нескольких разных форматах, то стандартизация формата для таких значений данных может оказаться сложной. Например, предположим, что данные в столбце `Customer.Phonenumber` хранятся в 15 различных форматах; в связи с этим потребуется код, способный распознавать все эти форматы, а затем преобразовывать данные в каждой строке в стандартный формат. При благоприятном стечении обстоятельств код, предназначенный для такого преобразования, должен уже присутствовать в одной или нескольких внешних программах.

Процедура обновления схемы

Прежде чем применять операцию “Введение общего формата” к столбцу, необходимо определить, каким должен быть стандартный формат. Иными словами, должен быть принят “официально утвержденный” формат для значений данных. Формат данных может определяться существующим или новым стандартом, согласованным со всеми заинтересованными лицами, в том числе отвечающими за производственную деятельность, но в любом случае формат должен быть утвержден тем лицом (лицами), в интересах которого разрабатывается проект. После этого необходимо выяснить, к какому столбцу (столбцам) должен применяться этот формат. Операцию “Введение общего формата” следует проводить применительно лишь к одному столбцу, поскольку меньшие операции рефакторинга всегда проще вначале осуществить, а затем развернуть на производстве. Возможно также, что эту операцию потребуется провести несколько раз для обеспечения согласованности форматов во всей базе данных.

Процедура переноса данных

На первом шаге необходимо выявить все возможные форматы, которые используются в настоящее время в рассматриваемом столбце, чтобы можно было определить, какие действия должны быть выполнены в коде переноса данных; для этого чаще всего достаточно воспользоваться простым оператором `SELECT`. А на втором шаге должен быть написан код преобразования существующих данных в стандартный формат. Такой код может быть подготовлен с использованием стандартного языка манипулирования данными (Data Manipulation Language — DML) в составе языка SQL; языка прикладного программирования, такого Java или C#, или инструментального средства ETL (Extract–Transform–Load — извлечь–преобразовать–загрузить). Если количество строк, требующих обновления, невелико, то достаточно применить простой сценарий SQL, который обновляет целевую таблицу (таблицы). А в тех случаях, когда требуется обновить большой объем данных или изменяется код в транзакционных таблицах, вместо этого следует применять операцию "Обновление данных" (с. 329).

В следующем коде приведен пример операторов DML, предназначенных для обновления данных в таблице `Customer`:

```
UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,'-','');

UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,' ','');

UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,'(','');

UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,')','');

UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,'+1','');

UPDATE Customer SET PhoneNumber =
      REPLACE(PhoneNumber,'.','');
```

Как показывает этот пример, в определенный момент времени обновляется формат только одного типа. Код внесения отдельных изменений можно также представить с помощью хранимой процедуры, как показано ниже.

```
UPDATE Address SET PhoneNumber = FormatPhoneNumber(PhoneNumber);
```

Процедура обновления программ доступа

Если применяется операция "Введение общего формата", то должны быть исследованы указанные ниже характеристики внешних программ.

1. **Код унификации данных.** Внешние программы должны содержать программные средства, которые воспринимают данные в различных форматах и преобразуют их в формат, необходимый им для работы. Как уже было сказано, часть этого

существующего кода возможно также использовать в качестве основы для создания кода переноса данных.

2. **Код проверки.** Может возникнуть необходимость откорректировать исходный код, предназначенный для проверки значений атрибутов данных. Например, код, в котором осуществляется поиск данных в определенном формате, такой как `Phonenumber = 'NNN-NNN-NNNN'`, должен быть откорректирован с учетом использования нового формата данных.
3. **Код вывода данных.** Код поддержки пользовательского интерфейса часто включает программные средства вывода элементов данных в определенном формате, который часто отличается от используемого для хранения данных (например, хранение осуществляется в формате `NNNNNNNNNN`, а отображение — в формате `(NNN) NNN-NNNN`). К этому относятся программные средства вывода в экранные формы и в отчеты.
4. **Проверочные данные.** После проведения этой операции рефакторинга необходимо изменить состав проверочных данных или откорректировать код выработки проверочных данных в соответствии с новым стандартным форматом данных. Может также возникнуть необходимость ввести новые тестовые примеры для проверки результатов обработки строк данных, представленных в недопустимом формате.

Операция рефакторинга “Введение заданного по умолчанию значения”

Эта операция позволяет предусмотреть в базе данных предоставление заданного по умолчанию значения для существующего столбца таблицы (рис. 7.10).

Рис. 7.10. Представление заданного по умолчанию значения для столбца `Customer.Status`

Обоснование

Часто возникает необходимость, чтобы для некоторого столбца было предусмотрено заданное по умолчанию значение, которое вводится в столбец при добавлении к таблице новой строки. Но в операторах вставки может быть не всегда предусмотрено заполнение этого столбца, чаще всего потому, что столбец был добавлен после написания первоначального оператора вставки, или просто в связи с тем, что в прикладном коде, содержащем оператор вставки, этот столбец не требуется. Вообще говоря, авторы обнаружили, что операция "Введение заданного по умолчанию значения" становится применимой, если требуется сделать в дальнейшем рассматриваемый столбец не допускающим NULL-значения (см. описание операции рефакторинга "Преобразование столбца в недопускающий NULL-значения" базы данных на с. 230).

Потенциальные преимущества и недостатки

Рассматривая возможность выполнения этой операции рефакторинга, необходимо учитывать несколько описанных ниже потенциальных затруднений.

- **Выявление действительно необходимого значения, применяемого по умолчанию, может оказаться сложным.** Если к одной и той же базе данных имеют совместный доступ многочисленные приложения, то в этих приложениях могут применяться разные заданные по умолчанию значения для одного и того же столбца, часто по весомым основаниям. Еще один вариант состоит в том, что другие заинтересованные лица, занимающиеся эксплуатацией приложений, просто не могут согласовать между собой единственное применимое значение; в таком случае необходимо обеспечить тесное сотрудничество с ними, чтобы договориться о выборе правильного значения.
- **Возникновение непредвиденных побочных эффектов.** Некоторые приложения могут быть разработаны на основании предположения, что NULL-значения в некотором столбце фактически имеют какой-то смысл. Поэтому подобные приложения будут обнаруживать иное поведение после того, как столбцы во вновь введенных строках, которые должны были прежде иметь NULL-значения, теперь их не содержат.
- **Наличие неоднозначного контекста.** Если некоторый столбец не используется в приложении, то при анализе применяемого по умолчанию значения может возникнуть непонимание со стороны группы разработчиков в отношении того, для чего предназначен этот столбец.

Процедура обновления схемы

Эта операция рефакторинга является одношаговой. При ее осуществлении достаточно воспользоваться командой `ALTER TABLE` языка SQL и определить заданное по умолчанию значение для столбца. При желании может быть дополнительно указана дата проведения этой операции рефакторинга, чтобы все заинтересованные лица могли знать, когда в схему было впервые введено рассматриваемое значение, заданное по умолчанию. В следующем коде показано, как ввести заданное по умолчанию значение для столбца `Customer.Status`, в соответствии с тем, что приведено на рис. 7.10:

```
ALTER TABLE Customer MODIFY Status DEFAULT 'NEW';

COMMENT ON Customer.Status 'Default value of NEW will be inserted when
there is no data present on the insert as of June 14 2006';
```

Процедура переноса данных

Существующие строки могут уже иметь NULL-значения в соответствующем столбце, поэтому в результате ввода в действие заданного по умолчанию значения эти строки не будут обновлены автоматически. Кроме того, в некоторых строках могут также присутствовать недействительные значения. Необходимо проверить, какие значения содержатся в столбце, чтобы определить, должны ли быть проведены какие-либо обновления; для этого может оказаться достаточным всего лишь изучение списка уникальных значений. Может также потребоваться в случае необходимости написать сценарий обработки рассматриваемой таблицы, который обеспечивает вставку заданных по умолчанию значений в подобные строки.

Процедура обновления программ доступа

На первый взгляд создается впечатление, что программы доступа вряд ли будут испытывать отрицательное влияние после перехода к применению значений по умолчанию, но такое впечатление может оказаться обманчивым. Потенциальные проблемы, которые могут возникнуть при осуществлении этой операции рефакторинга, включают описанные ниже.

1. **В результате ввода в действие нового значения может быть нарушено разнообразие.** Например, допустим, что в некотором программном классе принято предположение, что значение цвета, содержащееся в столбце, является либо красным, либо зеленым, либо синим, но теперь применяемое по умолчанию значение цвета определено как “желтый”.
2. **Предусмотренные по умолчанию значения вводятся с помощью существующего кода.** Может оказаться так, что существует внешний исходный код, в котором проверяется наличие NULL-значений и ввод заданных по умолчанию значений осуществляется программным путем. Этот код должен быть удален.
3. **В существующем исходном коде допускается возможность применения другого заданного по умолчанию значения.** Например, предположим, что в существующем коде предусмотрен поиск заданного по умолчанию значения “цвет неизвестен”, который был установлен программным путем в прошлом, а в случае его обнаружения пользователям предоставляется возможность задать значение цвета. А теперь по умолчанию применяется значение “желтый”, поэтому код, в котором определено другое условие, никогда не будет вызываться.

Необходимо всесторонне проанализировать программы доступа, а затем обновить их должным образом, и только после этого переходить к использованию в столбце заданного по умолчанию значения.

Операция рефакторинга “Преобразование столбца в недопускающий NULL-значения”

Эта операция позволяет откорректировать существующий столбец таким образом, чтобы в нем больше не допускалось применение каких-либо NULL-значений (рис. 7.11).

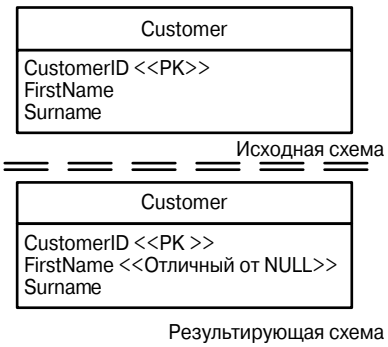


Рис. 7.11. Преобразование столбца `Customer.FirstName` с тем, чтобы в нем не допускалось использование NULL-значений

Обоснование

Необходимость в применении операции “Преобразование столбца в недопускающий NULL-значения” может быть обусловлено двумя причинами. Во-первых, может потребоваться обеспечить соблюдение бизнес-правил на уровне базы данных, для того чтобы в любом приложении, обновляющем рассматриваемый столбец, приходилось задавать для него определенное значение. Во-вторых, может потребоваться удалить из приложений повторяющийся программный код, в котором реализована проверка на отличие от `NULL`; ведь если вставка NULL-значений не допускается, то с такими значениями никогда не придется сталкиваться.

Потенциальные преимущества и недостатки

В любой внешней программе, которая обновляет строки в соответствующей таблице, необходимо будет задавать значение для рассматриваемого столбца, тогда как некоторые программы в настоящее время могут быть разработаны с учетом предположения, что столбец допускает применение NULL-значений, и поэтому в них не предусмотрено использование значений, отличных от `NULL`. При выполнении каждой операции обновления или вставки необходимо предоставить какое-то значение, а из этого следует, что должны быть откорректированы внешние программы и (или) внесены исправления в саму базу данных, чтобы в столбце всегда было предусмотрено допустимое значение. Авторы пришли к выводу, что один из удобных методов состоит в присваивании заданного по умолчанию значения путем выполнения операции “Введение заданного по умолчанию значения” (с. 213) применительно к рассматриваемому столбцу.

Процедура обновления схемы

Как показано на рис. 7.11, для выполнения операции “Преобразование столбца в недопускающий NULL-значения” достаточно ввести ограничение `NOT NULL` на рассматриваемом столбце. Пример, приведенный в следующем коде, показывает, что такая доработка выполняется с помощью команды `ALTER TABLE` языка SQL:

```
ALTER TABLE Customer
  MODIFY FirstName NOT NULL;
```

Применение простого способа изображения моделей

На рис. 7.11 демонстрируется любопытная проблема, касающаяся принятого стиля отображения; на этом рисунке на исходной схеме не показано, что столбец `FirstName` допускает применение NULL-значений. Предположение, согласно которому столбцы, не являющиеся частью первичного ключа, допускают применение NULL-значений, используется очень часто, поэтому, если бы для большинства столбцов было указано стереотипное значение `<<nullable>>` (допускающий применение NULL-значений), возникло бы лишь ненужное загромождение схемы. Схемы становятся более удобными для чтения, если столбцы отображаются без указанного стереотипного значения.

Процедура переноса данных

Может потребоваться унифицировать существующие данные, поскольку столбец нельзя обозначить как недопускающий NULL-значения, если в этом столбце присутствуют строки, содержащие NULL-значения. Для устранения этой проблемы необходимо написать сценарий, который обновляет строки, содержащие NULL-значения, вводя более приемлемое значение.

В следующем коде показано, как унифицировать существующие данные, чтобы обеспечить осуществление изменения, показанного на рис. 7.11. На начальном этапе необходимо убедиться в том, что столбец `FirstName` не содержит NULL-значений; если таковые в нем имеются, то следует обновить данные в таблице:

```
SELECT count(FirstName) FROM Customer
  WHERE
    FirstName IS NULL;
```

Обнаружив такие строки, где в столбце `Customer.FirstName` содержится NULL-значение, необходимо продолжить работу, применить какой-то алгоритм корректировки и убедиться в том, что столбец `Customer.FirstName` больше не содержит NULL-значений. В данном примере задается значение `Customer.FirstName`, равное `'???'`, которое указывает, что необходимо обновить соответствующую строку. Выбранная нами стратегия была полностью согласована с другими лицами, заинтересованными в разработке проекта, поскольку данные, подвергающиеся изменению, важны для предприятия:

```
UPDATE Customer SET FirstName='???'
  WHERE
    FirstName IS NULL;
```

Процедура обновления программ доступа

Необходимо подвергнуть рефакторингу все внешние программы, получающие в настоящее время доступ к столбцу `Customer.FirstName`, чтобы эти программы предоставляли приемлемое значение при каждой модификации с их помощью любой строки в таблице. Кроме того, внешние программы должны обнаруживать, а затем обрабатывать любые новые исключения, которые активизируются базой данных.

Если невозможно со всей определенностью выявить все ситуации, в которых используется рассматриваемый столбец, а также отсутствует возможность внести изменения применительно ко всем экземплярам программ, в которых он используется, то можно применить операцию рефакторинга базы данных "Введение заданного по умолчанию значения" (с. 213), чтобы значение, заданное по умолчанию, предоставляла база данных, если приложение не предоставляет никакого значения. Такая стратегия рефакторинга может стать промежуточной, а по истечении достаточного времени, когда появится полная уверенность в том, что все программы доступа откорректированы, или, по крайней мере, после того, как все будет готово для быстрой модификации той горстки программ доступа, которые еще не были откорректированы, можно применить операцию "Удаление значения, заданного по умолчанию" (с. 203) и таким образом повысить производительность.

В следующем коде показано состояние до и после проведения операции рефакторинга "Преобразование столбца в недопускающий NULL-значения". В рассматриваемом примере было решено активизировать исключение при обнаружении NULL-значения:

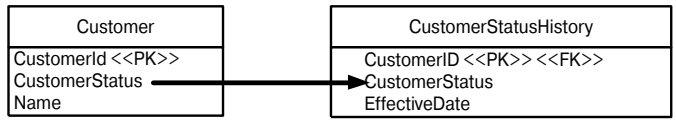
```
// Код до рефакторинга
stmt = conn.prepare(
  "INSERT INTO Customer "+
  "(CustomerID,FirstName,Surname) "+
  "VALUES (?,?,?,)");
stmt.setLong(1,customerId);
stmt.setString(2,firstName);
stmt.setString(3,surname);
stmt.executeUpdate();
}

// Код после рефакторинга
if (firstName == null) {
  throw new CustomerFirstNameCannotBeNullException();
};
stmt = conn.prepare(
  "INSERT INTO Customer "+
  "(CustomerID,FirstName,Surname) "+
  "VALUES (?,?,?,)");
stmt.setLong(1,customerId);
stmt.setString(2,firstName);
stmt.setString(3,surname);
stmt.executeUpdate();
}
```

Операция рефакторинга “Перемещение данных”

Эта операция предназначена для перемещения данных, содержащихся в таблице, либо полностью, либо ограничиваясь подмножеством столбцов, в другую существующую таблицу (рис. 7.12).

Схема



Значения данных до рефакторинга

Customer		
CustomerID	CustomerStatus	Name
1	Active	Allen, B.
2	Pending	Jordan, H.
3	Active	Wayne, B.
4	Active	Parker, P.
5	In Arrears	Banner, B.

Значения данных после рефакторинга

Customer		
CustomerID	CustomerStatus	Name
1	NULL	Allen, B.
2	NULL	Jordan, H.
3	NULL	Wayne, B.
4	NULL	Parker, P.
5	NULL	Banner, B.

CustomerStatusHistory		
CustomerID	CustomerStatus	EffectiveDate
1	Active	2006-04-09
2	Pending	2006-04-09
3	Active	2006-04-09
4	Active	2006-04-09
5	In Arrears	2006-04-09

Рис. 7.12. Перемещение данных о состоянии из таблицы `Customer` в таблицу `CustomerStatusHistory`

Обоснование

Необходимость в использовании операции “Перемещение данных” обычно становится результатом структурных изменений в проекте таблицы; но, как всегда, нюансы применения этой операции рефакторинга зависят от ситуации. В частности, применение операции “Перемещение данных” может потребоваться в результате осуществления описанных ниже действий.

- **Переименование столбца.** При переименовании столбца осуществляется процесс, в котором вначале вводится новый столбец с новым именем, затем исходные данные перемещаются в новый столбец, после чего исходный столбец удаляется с помощью операции “Удаление столбца” (с. 112).
- **Разбиение таблицы по вертикали.** После применения операции “Разбиение таблицы” (с. 177) для реорганизации существующей таблицы с разбивкой ее по верти-

кали необходимо переместить данные из исходной таблицы в новые таблицы, созданные в результате ее разбиения.

- **Разбиение таблицы по горизонтали.** Иногда выделенный по горизонтали срез данных перемещается из одной таблицы в другую таблицу с такой же структурой, поскольку исходная таблица увеличилась до такой степени, что производительность снизилась. Например, данные обо всех европейских клиентах могут быть перемещены из таблицы `Customer` в таблицу `EuropeanCustomer`. Такое разбиение по горизонтали может стать первым шагом к формированию иерархии наследования (в которой, например, таблица `EuropeanCustomer` наследует свои свойства от таблицы `Customer`) в базе данных и (или) может быть связано с тем, что необходимо добавить определенные столбцы, характерные только для европейских клиентов.
- **Разбиение столбца.** После проведения операции "Разбиение столбца" (с. 172) возникает необходимость переместить данные из исходного столбца в новый столбец (столбцы).
- **Слияние таблиц или столбцов.** После применения операции "Слияние таблиц" (с. 133) или "Слияние столбцов" (с. 130) возникает необходимость переместить данные из исходного столбца (столбцов) в целевой столбец (столбцы).
- **Консолидация данных, не предусматривающая структурных изменений.** Данные часто хранятся в нескольких местоположениях. Например, может оказаться, что данные о клиентах хранятся в разных таблицах, при этом в отдельных таблицах находятся данные о клиентах из Канады, а также о клиентах, проживающих в различных регионах Соединенных Штатов, но все эти таблицы имеют одну и ту же основную структуру. А в результате реорганизации корпорации данные, относящиеся к провинциям Британская Колумбия и Альберта, перемещаются из таблицы `CanadianCustomer` в таблицу `WestCoastCustomer`, что соответствуют изменению прав владения данными в этой организации.
- **Перемещение данных перед удалением таблицы или столбца.** Перед применением операции "Удаление таблицы" (с. 117) или "Удаление столбца" (с. 112) может потребоваться предварительно применить операцию "Перемещение данных", если все еще остается необходимость в использовании некоторых или всех данных, хранимых в таблице или столбце.

Потенциальные преимущества и недостатки

Задача перемещения данных из одного столбца в другой всегда может быть связана с какими-то сложностями, но становится особенно затруднительной, если количество перемещаемых строк исчисляется миллионами. Во время такого перемещения приложения, получающие доступ к данным, могут испытывать отрицательное влияние; мало того, во время перемещения может потребоваться заблокировать данные, в результате чего снизится общая производительность и доступность данных для приложений, поскольку в таком случае приложения не смогут получить доступ к данным в течение всего времени перемещения.

Процедура обновления схемы

Чтобы выполнить операцию “Перемещение данных”, необходимо вначале выявить все перемещаемые данные и определить любые зависимости, в которых они участвуют. При этом должен быть найден ответ на вопрос о том, следует ли удалить соответствующие строки в другой таблице, заменить NULL-значением, или обнулить значение в соответствующем столбце, или оставить соответствующее значение неизменным. Кроме того, необходимо выяснить, перемещается ли вся строка или только некоторый столбец (столбцы).

Не менее важно то, что должен быть определен адресат данных. При этом необходимо выяснить, куда перемещаются данные, является ли местом назначения другая таблица или таблицы, осуществляется ли преобразование данных во время перемещения. При перемещении строк необходимо обеспечить, чтобы все зависимые таблицы, в которых имеется внешний ключ, ссылающийся на таблицу, в которой находятся перемещаемые данные, теперь ссылались на таблицу, представляющую собой место назначения для перемещения.

Процедура переноса данных

Если требуется переместить небольшой объем данных, то, по-видимому, достаточно применить простой сценарий SQL, который вставляет исходные данные в целевом местоположении, а затем удаляет исходные данные. Если же перемещению подлежат значительные объемы данных, то должен быть принят более сложный подход, поскольку для перемещения таких данных может потребоваться много времени. В частности, иногда бывает целесообразно экспортировать исходные данные, а затем импортировать их в целевом местоположении. Может также оказаться оправданным использование таких утилит базы данных, как SQLLDR в СУБД Oracle или массовый загрузчик.

На рис. 7.12 показано, как переместить данные из столбца `Customer.Status` в таблицу `CustomerStatusHistory`, чтобы иметь возможность отслеживать статус клиента в течение определенного периода времени. В следующем коде приведены операторы DML, предназначенные для перемещения данных из столбца `Customer.Status` в таблицу `CustomerStatusHistory`. Вначале осуществляется вставка данных в таблицу `CustomerStatusHistory` из таблицы `Customer`, а затем обновляется столбец `Customer.Status` и ему присваются NULL-значения:

```
INSERT INTO CustomerStatusHistory (CustomerId,Status,EffectiveDate)
SELECT CustomerId,Status,Sysdate FROM Customer;
UPDATE Customer SET Status = NULL;
```

Процедура обновления программ доступа

Если операция “Перемещение данных” применяется как следствие применения другой операции рефакторинга базы данных, то в соответствии с этой операцией рефакторинга должны быть обновлены приложения. Но если операция “Перемещение данных” осуществляется для реорганизации данных в пределах существующей схемы, то, возможно, потребует обновления внешний прикладной код. Например, может возникнуть необходимость переопределить конструкции `SELECT`, чтобы в них был предусмотрен доступ к перемещенным данным, получаемым из новой исходной таблицы (таблиц). В следующем фрагменте кода показано, как изменяется метод `getCustomerStatus()`:

```
// Код до рефакторинга
public String getCustomerStatus(Customer customer) {
  return customer.getStatus();
}

// Код после рефакторинга
public String getCustomerStatus(Customer customer) throws SQLException
{
  stmt.prepare(
    "SELECT Status "+
    "FROM CustomerStatusHistory " +
    "WHERE " +
    "CustomerId = ? "+
    "AND EffectiveDate < TRUNC(SYSDATE) "+
    "ORDER BY EffectiveDate DESC");
    stmt.setLong(1,customer.getCustomerId);
  ResultSet rs = stmt.execute();
  if (rs.next()) {
    return rs.getString("Status");
  }
  throw new CustomerStatusNotFoundInHistoryException();
}
```

Операция рефакторинга "Замена кодового обозначения типа флажками свойств"

Эта операция предназначена для замены столбца с кодовыми обозначениями отдельными флажками свойств, обычно реализуемыми как булевы столбцы, в том же столбце таблицы (рис. 7.13).

Следует отметить, что некоторые программные продукты баз данных поддерживают собственные булевы столбцы. Другие же программные продукты баз данных позволяют использовать в качестве булевых значений альтернативные типы данных (например, `NUMBER(1)` в Oracle, где значение `1` соответствует `TRUE`, а значение `0` представляет `FALSE`).

Обоснование

Операция рефакторинга "Замена кодового обозначения типа флажками свойств" используется для осуществления описанных ниже действий.

- **Денормализация в целях повышения производительности.** Иногда повышению производительности способствует применение отдельного столбца для каждого экземпляра кода типа. Например, предположим, что в столбце `Address.Type` могут находиться значения `Home` и `Business`; этот столбец можно заменить столбцами флажков свойств `isHome` и `isBusiness`. Благодаря этому эффективность поиска увеличивается, поскольку операция сравнения двух булевых значений выполняется быстрее, чем сравнение двух строковых значений.

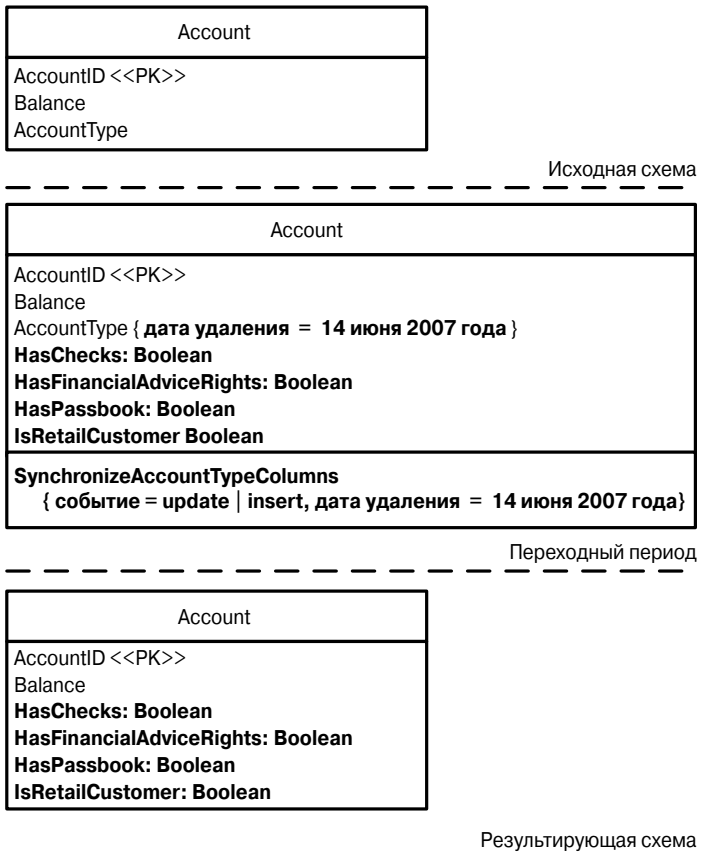


Рис. 7.13. Замена столбца `AccountType` с кодовыми обозначениями флажками свойств

- **Упрощение операции выборки.** Столбцы с кодовыми обозначениями типов применяются успешно, если типы являются взаимоисключающими, в противном случае при осуществлении поиска в таких столбцах приходится сталкиваться с проблемами. Например, предположим, что адрес может быть либо домашним, либо служебным. При использовании подхода, основанного на обработке кодовых обозначений типа, может потребоваться, например, чтобы в стобце `Address.Type` были представлены значения `Home`, `Business` и `HomeAndBusiness`. Чтобы получить список всех служебных адресов, необходимо выполнить примерно такой запрос: `SELECT * FROM Address WHERE Type = "Business" OR Type = "HomeAndBusiness"`. Вполне очевидно, что формулировку этого запроса пришлось бы корректировать после добавления каждого нового варианта типа адреса, например домашнего адреса на время отпуска, но, допустим, с учетом того, что для связи с интересующим нас лицом в этот период можно было бы также воспользоваться служебным адресом. Если же используется подход, в основе которого лежат столбцы с флажками свойств, то подобный запрос могут выглядеть примерно как `SELECT * FROM Address`

`WHERE isBusiness = TRUE`. Этот запрос является несложным и не требует корректировки после добавления нового типа адреса.

- **Обеспечение независимости приложений от кодовых обозначений типа.** Если столбец `Account.AccountType`, показанный на рис. 7.13, используется в нескольких приложениях, то внесение изменений в содержащиеся в нем значения становится затруднительным, поскольку в большинстве приложений соответствующие кодовые обозначения обычно принято задавать с помощью программных конструкций. Если же такие столбцы с кодовыми обозначениями типа заменяются столбцами с флажками свойств, то в приложениях достаточно предусмотреть проверку стандартных значений `TRUE` и `FALSE`. После того как столбец с кодовым обозначением типа в приложениях будет связан с именем столбца и значениями этого столбца, достаточно лишь установить в приложениях связь столбцов с флажками свойств и имен столбцов.

Потенциальные преимущества и недостатки

После каждого добавления обозначения нового типа приходится изменять структуру таблицы. Например, если необходимо ввести дополнительно тип депозитного счета денежного рынка, то достаточно добавить столбец `isMoneyMarket` в таблицу `Account`. Но со временем выполнение подобного действия становится нежелательным, поскольку таблицы с большим количеством столбцов труднее поддаются пониманию по сравнению с таблицами, имеющими меньшее количество столбцов. Но добавить столбец, независимый от остальной части столбцов, очень легко. При использовании подхода, основанного на столбце с кодовыми обозначениями типа, рассматриваемый столбец связан со всеми приложениями, получающими к нему доступ, поэтому добавление нового кодового обозначения потенциально становится причиной воздействия на все эти взаимосвязанные программы.

Процедура обновления схемы

Чтобы иметь возможность применить операцию “Замена кодового обозначения типа флажками свойств” к таблице базы данных, необходимо выполнить описанные ниже действия.

1. **Выявить кодовое обозначение типа, подлежащее замене.** Необходимо найти столбец с кодовыми обозначениями типа, подлежаший замене; в рассматриваемом примере таковым является `Account.AccountType`. Необходимо также получить сведения обо всех экземплярах кодовых обозначений типа, которые должны быть заменены; например, если требуется заменить кодовые обозначения `AddressType`, то необходимо найти все возможные типы адресов и заменить все экземпляры обозначения `AddressType`.
2. **Ввести столбцы с флажками свойств.** После выявления всех экземпляров кодовых обозначений типа, которые подлежат замене, и определения их количества необходимо добавить такое же количество столбцов в таблицу с помощью операции “Введение нового столбца” (с. 321). Как показано на рис. 7.13, требуется дополнительно ввести в таблицу `Account` столбцы `HasChecks`, `HasFinancialAdviceRights`, `HasPassbook` и `IsRetailCustomer`.

3. **Удалить столбец с кодовыми обозначениями типа.** После того как все кодовые обозначения типа будут заменены флажками свойств, необходимо удалить столбец с кодовыми обозначениями типа, используя операцию “Удаление столбца” (с. 112).

На рис. 7.13 показано, как модифицировать столбец `Account.AccountType`, чтобы в нем использовались столбцы с флажками типа для каждого экземпляра данных `AccountType`. В рассматриваемом примере имеются четыре различных обозначения типа счета, которые должны быть преобразованы в столбцы с флажками типа, получившие название `HasChecks`, `HasFinancialAdviceRights`, `HasPassbook` и `isRetailCustomer`. В следующем коде SQL показано, как добавить четыре столбца с флажками типа в таблицу `Account` и ввести в действие триггер синхронизации на время переходного периода, а также приведены операторы, предназначенные для удаления исходного столбца, и дано определение триггера, который будет применяться после окончания переходного периода:

```
ALTER TABLE Account ADD COLUMN HasChecks Boolean;
COMMENT ON Account.HasChecks "Replaces AccountType of Checks"

ALTER TABLE Account ADD COLUMN HasFinancialAdviceRights Boolean;
COMMENT ON Account.HasFinancialAdviceRights "Replaces AccountType of
FinancialAdviceRights"

ALTER TABLE Account ADD COLUMN HasPassbook Boolean;
COMMENT ON Account.HasPassbook "Replaces AccountType of Passbook"

ALTER TABLE Account ADD COLUMN isRetailCustomer Boolean;
COMMENT ON Account. isRetailCustomer "Replaces AccountType of
RetailCustomer"

CREATE OR REPLACE TRIGGER SynchronizeAccountTypeColumns
BEFORE INSERT OR UPDATE
ON ACCOUNT
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF (:NEW.HasChecks IS NULL AND :NEW.AccountType = 'CHECKS') THEN
    :NEW.HasChecks := TRUE;
  END IF;
  IF (:NEW.HasFinancialAdviceRights IS NULL AND :NEW.AccountType =
'FINANCIALADVICERIGHTS') THEN
    :NEW.HasFinancialAdviceRights := TRUE;
  END IF;
  IF (:NEW.HasPassbook IS NULL AND :NEW.AccountType = 'PASSBOOK') THEN
    NEW.HasPassbook := TRUE;
  END IF;
  IF (:NEW.isRetailCustomer IS NULL AND :NEW.AccountType =
'RETAILCUSTOMER') THEN
    :NEW.isRetailCustomer := TRUE;
  END IF;
END;
/
```

```
-- 14 июня 2007 года
ALTER TABLE Account DROP COLUMN AccountType;
DROP TRIGGER SynchronizeAccountTypeColumns;
```

Процедура переноса данных

Необходимо подготовить сценарии переноса данных для обновления флажков свойств с учетом кодовых обозначений типа; для этого можно использовать операцию "Обновление данных" (с. 329). Во время переходного периода, в течение которого применяются и столбец с кодовыми обозначениями типа, и столбцы с флажками типа, необходимо поддерживать синхронизацию столбцов, для того чтобы приложения, в которых используются эти данные, получали согласованную информацию. Этой цели можно достичь с использованием триггеров базы данных. Ниже приведены операторы SQL, которые показывают, как обновить существующие данные в таблице `Account` (дополнительные сведения приведены в описании операции "Обновление данных" на с. 329).

```
UPDATE Account SET HasChecks = TRUE
  WHERE AccountType = 'CHECKS';
UPDATE Account SET HasChecks = FALSE
  WHERE HasChecks != TRUE;

UPDATE Account SET HasFinancialAdviceRights = TRUE
  WHERE AccountType = 'FINANCIALADVICERIGHTS';
UPDATE Account SET HasFinancialAdviceRights = FALSE
  WHERE HasFinancialAdviceRights != TRUE;

UPDATE Account SET HasPassbook = TRUE
  WHERE Accounttype = 'PASSBOOK';
UPDATE Account SET HasPassbook = FALSE
  WHERE HasPassbook != TRUE;

UPDATE Account SET isRetailCustomer = TRUE
  WHERE Accounttype ='RETAILCUSTOMER';
UPDATE Account SET isRetailCustomer = FALSE
  WHERE isRetailCustomer!= TRUE;
```

Процедура обновления программ доступа

При использовании операции "Замена кодового обозначения типа флажками свойств" необходимо обновить внешние программы в двух следующих аспектах. Во-первых, должны быть откорректированы операторы SQL (или метаданные), которые применяются для сохранения, удаления и выборки данных из рассматриваемого столбца, чтобы они могли использовать отдельные столбцы с флажками типа, а не столбец `AccountType`. Например, если в настоящее время применяется оператор `SELECT * FROM Account WHERE AccountType = 'XXXX'`, этот оператор SQL необходимо заменить оператором `SELECT * FROM Account WHERE isXXXX = TRUE`. Аналогичным образом, код программы должен быть заменен так, чтобы в нем во время выполнения операций вставки или обновления предусматривалось обновление столбцов с флажками типа, а не столбца `AccountType`. Во-вторых, может потребовать обновления прикладной

код, который работает со столбцом. Например, должны быть откорректированы операторы сравнения, такие как `Customer.AddressType = 'Home'`, в данном случае для работы со столбцом `isHome`.

В следующем примере показано, как изменяется класс `Account` после замены кодовых обозначений типа счета флажками свойств:

```
// Код до рефакторинга
public class Account {

private Long accountID;
private BigDecimal balance;
private String accountType;
private Boolean FALSE = Boolean.FALSE;
private Boolean TRUE = Boolean.TRUE;

public Long getAccountID() {
  return accountID;
}

public BigDecimal getBalance() {
  return balance;
}

public Boolean HasChecks() {
  return accountType.equals("CHECKS");
}

public Boolean HasFinancialAdviceRights() {
  return accountType.equals("FINANCIALADVICERIGHTS");
}

public Boolean HasPassBook() {
  return accountType.equals("PASSBOOK");
}

public Boolean IsRetailCustomer() {
  return accountType.equals("RETAILCUSTOMER");
}
}

// Код после рефакторинга
public class Account {

private Long accountID;
private BigDecimal balance;
private Boolean HasChecks;
private Boolean HasFinancialAdviceRights;
private Boolean HasPassBook;
private Boolean IsRetailCustomer;

public Long getAccountID() {
  return accountID;
}
```

```
public BigDecimal getBalance() {
  return balance;
}

public Boolean HasChecks() {
  return HasChecks;
}

public Boolean HasFinancialAdviceRights() {
  return HasFinancialAdviceRights;
}

public Boolean HasPassBook() {
  return HasPassBook;
}

public Boolean IsRetailCustomer() {
  return IsRetailCustomer;
}
}
```

Глава 8

Операции рефакторинга ссылочной целостности

Операции рефакторинга ссылочной целостности представляют собой модификации, проведение которых гарантирует, что строки, на которые имеется ссылка, будут существовать в другой таблице и (или) не нужные больше строки будут удаляться должным образом. Операции рефакторинга ссылочной целостности перечислены ниже.

- Операция рефакторинга “Добавление ограничения внешнего ключа”.
- Операция рефакторинга “Добавление триггера для вычисляемого столбца”.
- Операция рефакторинга “Уничтожение ограничения внешнего ключа”.
- Операция рефакторинга “Введение каскадного удаления”.
- Операция рефакторинга “Введение физического удаления”.
- Операция рефакторинга “Введение программного удаления”.
- Операция рефакторинга “Введение триггера для накопления исторических данных”.

Операция рефакторинга “Добавление ограничения внешнего ключа”

Эта операция позволяет ввести в существующую таблицу ограничение внешнего ключа, которое принудительно устанавливает связь с другой таблицей (рис. 8.1).

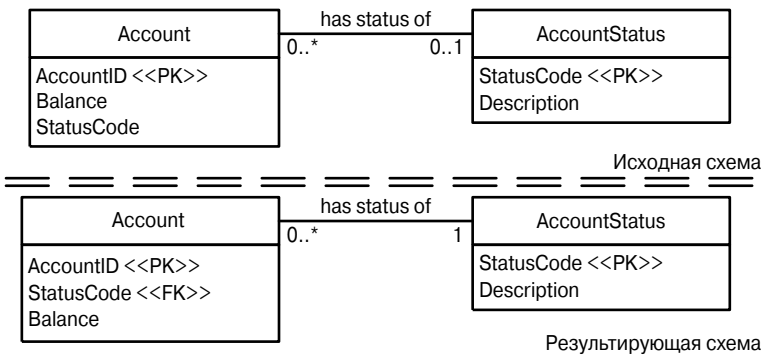


Рис. 8.1. Ввод в действие ограничения внешнего ключа на столбце `Account.AccountStatus`

Обоснование

Необходимость в применении операции "Добавление ограничения внешнего ключа" в основном обусловлена тем, что должны быть принудительно введены в действие зависимости между данными на уровне базы данных, гарантирующие, что база данных будет поддерживать некоторые бизнес-правила обеспечения ссылочной целостности (Referential Integrity — RI), предотвращающие возможность сохранения в базе данных недопустимых данных. Это особенно важно, если доступ к одной и той же базе данных имеет несколько приложений, поскольку нельзя рассчитывать на то, что эти приложения будут взаимно согласовывать выполнение правил целостности данных.

Потенциальные преимущества и недостатки

В результате ввода в действие ограничений внешнего ключа в базе данных производительность снижается, поскольку при каждом обновлении строки в исходной таблице приходится проверять наличие связанной с ней строки в таблице внешнего ключа. Кроме того, после ввода в действие ограничения внешнего ключа применительно к базе данных возникает необходимость следить за соблюдением правильной последовательности операций вставки, обновления и удаления. Например, как показано на рис. 8.1, невозможно добавить строку в таблицу `Account`, если отсутствует соответствующая строка в таблице `AccountStatus`. Из этого следует, что в приложении, вернее, в тех программных средствах, которые обеспечивают взаимодействие с базой данных, приходится учитывать наличие зависимостей между таблицами в базе данных. К счастью, многие базы данных позволяют проверять соблюдение ограничений базы данных во время фиксации данных, что дает возможность выполнять операции вставки/обновления или удаления в любом порядке, при условии, что действия по обеспечению целостности данных осуществляются во время фиксации. Средства подобного рода способствуют упрощению разработки и, в частности, в большей степени стимулируют применение ограничений внешнего ключа.

Процедура обновления схемы

Как показано на рис. 8.1, для обновления схемы в целях добавления ограничения внешнего ключа необходимо выполнить описанные ниже действия.

1. **Выбрать стратегию проверки ограничений.** В зависимости от применяемого программного продукта баз данных могут поддерживаться один или два способа предписания ограничений внешнего ключа. Первый способ, предусматривающий немедленную проверку, основан на том, что проверка ограничения внешнего ключа выполняется при вставке/обновлении или удалении данных из таблицы. Преимуществом метода немедленной проверки является то, что он позволяет быстрее обнаружить нарушение последовательности выполнения операций, но при его использовании приходится учитывать порядок внесения изменений в базу данных (вставок, обновлений и удалений). Второй способ, предусматривающий отложенную проверку ограничений, предназначен для проверки ограничения внешнего ключа при выдаче команды фиксации транзакции из приложения. Этот способ является более удобным, поскольку не требует соблюдения определенной последовательности внесения изменений в базу данных в связи с тем, что проверка ограничений выполняется во время фиксации. Кроме того, этот подход

позволяет кэшировать все незафиксированные объекты, а затем вносить связанные с ними изменения в базу данных в виде одного пакета; необходимо лишь обеспечить, чтобы ко времени фиксации этих изменений все предыдущие транзакции были полностью зафиксированы в базе данных. Но и в том и в другом случае база данных активизирует исключение сразу после обнаружения первого же нарушения ограничения внешнего ключа. (Хотя таких нарушений может быть несколько.)

2. **Создать ограничение внешнего ключа.** Ограничение внешнего ключа в базе данных создается с помощью конструкции `ADD CONSTRAINT` оператора `ALTER TABLE`. Ограничению базы данных следует присваивать имя с учетом соглашений об именовании объектов базы данных, поскольку это позволяет добиться определенности и обеспечить эффективную обработку сообщений об ошибках, поступающих из базы данных. Если применяется способ, предусматривающий проверку ограничений во время фиксации, то производительность может снизиться, поскольку база данных проверяет целостность данных во время фиксации, а это требует выполнения большого объема работы, если в таблицах содержатся миллионы строк.
3. **Ввести индекс для первичного ключа таблицы с внешним ключом (необязательно).** В базах данных для проверки допустимости данных, вводимых в дочернюю таблицу, используются операторы выборки, выполняемые на таблицах, указанных в ссылке. Это означает, что если, например, на столбце `AccountStatus.StatusCode` не задан индекс, то может обнаруживаться существенное снижение производительности, поэтому необходимо будет предусмотреть применение операции рефакторинга базы данных “Введение индекса” (с. 271). В результате создания подобного индекса увеличится производительность проверки ограничений, но уменьшится производительность операций обновления, вставки и удаления, в данном случае применительно к таблице `AccountStatus`, поскольку теперь база данных должна поддерживать еще один индекс.

В следующем примере кода показаны этапы добавления ограничения внешнего ключа к таблице. В этом примере ограничение создается таким образом, что проверка ограничения внешнего ключа осуществляется непосредственно после модификации данных:

```
ALTER TABLE Account
  ADD CONSTRAINT FK_Account_AccountStatus
  FOREIGN KEY (StatusCode)
  REFERENCES AccountStatus;
```

А в следующем примере ограничение внешнего ключа создается таким образом, что проверка ограничения внешнего ключа происходит во время фиксации:

```
ALTER TABLE Account
  ADD CONSTRAINT FK_Account_AccountStatus
  FOREIGN KEY (StatusCode)
  REFERENCES AccountStatus
  INITIALLY DEFERRED;
```

Процедура переноса данных

В ходе подготовки к вводу в действие нового ограничения внешнего ключа для таблицы может быть обнаружено, что необходимо вначале обновить существующие данные в базе данных. Как описано ниже, этот процесс является многошаговым.

1. **Обеспечить наличие данных, указанных в ссылке.** Необходимо прежде всего обеспечить, чтобы в таблице `AccountStatus` существовали строки, указанные в ссылке; проанализировать существующие данные и в таблице `Account`, и в таблице `AccountStatus`, чтобы определить, нет ли отсутствующих строк в таблице `AccountStatus`. Самый простой способ выполнения этой задачи состоит в том, чтобы сравнить количество строк в таблице `Account` с количеством строк, полученных в результате соединения таблиц `Account` и `AccountStatus`.
2. **Добиться того, чтобы в таблицу внешнего ключа были включены все требуемые строки.** Если количество строк в ссылающейся таблице и в таблице, указанной в ссылке, различается, это означает, что в таблице `AccountStatus` отсутствуют необходимые строки и (или) в таблице `Account.StatusCode` имеются неправильные значения. В таком случае, во-первых, необходимо создать список уникальных значений в столбце `Account.StatusCode` и сравнить его со списком значений из столбца `AccountStatus.StatusCode`. Если первый список содержит значения, которые являются действительными, но не присутствуют во втором списке, то необходимо обновить таблицу `AccountStatus`. Во-вторых, может оказаться так, что имеются еще некоторые допустимые значения, которых нет ни в одном списке, но они являются допустимыми в рассматриваемом домене данных конкретного делового предприятия. Для выявления этих значений необходимо действовать в тесном сотрудничестве со всеми лицами, заинтересованными в разработке текущего проекта; но еще лучше подождать, пока пользователям не потребуются строки с недостающими данными, а затем добавить необходимые строки.
3. **Обеспечить наличие в столбце внешнего ключа исходной таблицы допустимых значений.** Для этого следует обновить списки, полученные в результате выполнения предыдущего шага. Любые различия между списками теперь должны рассматриваться как свидетельство о том, что в столбце `Account.StatusCode` обнаруживаются неправильные или недостающие значения. Такие строки необходимо обновить соответствующим образом: либо с помощью автоматизированного сценария, который задает применяемое по умолчанию значение, либо вручную, обновляя отдельно каждое значение.
4. **Ввести значение по умолчанию для столбца внешнего ключа.** Дополнительно может потребоваться вставить с помощью базы данных значения, заданные по умолчанию, если внешние программы не будут предоставлять значение для столбца `Account.StatusCode`. См. описание операции рефакторинга базы данных “Введение заданного по умолчанию значения” (с. 213).

Применительно к примеру, показанному на рис. 8.1, необходимо обеспечить, чтобы перед вводом в действие ограничения внешнего ключа данные в таблицах соответствовали этому ограничению; в противном случае необходимо обновить эти данные. Предположим, что в таблице `Account` имеются такие строки, в которых состояние счета не зада-

но или задано такое значение состояния, для которого нет аналога в таблице Account-Status. В этой ситуации необходимо обновить столбец Account.Status, заменив подобные значения некоторым известным значением, которое существует в таблице AccountStatus:

```
UPDATE Account SET Status = 'DORMANT'
  WHERE
    Status NOT IN (SELECT StatusCode FROM AccountStatus)
    AND Status IS NOT NULL;
```

А в других случаях может потребоваться предусмотреть такую возможность, чтобы столбец Account.Status допускал наличие NULL-значений. В таком случае необходимо обновить столбец Account.Status, заменив NULL-значения известным значением, как показано ниже.

```
UPDATE Account SET Status = 'NEW'
  WHERE Status IS NULL;
```

Процедура обновления программ доступа

Необходимо выявить, а затем доработать все внешние программы, предназначенные для внесения изменений в данные таблицы, на которой введено ограничение внешнего ключа. При этом необходимо учитывать наличие описанных ниже проблем.

1. **Аналогичный код поддержки ссылочной целостности.** В некоторых внешних программах может быть реализовано бизнес-правило поддержки ссылочной целостности, которое теперь введено в действие с помощью ограничения внешнего ключа, поддерживаемого самой базой данных. Этот код следует удалить.
2. **Другой код поддержки ссылочной целостности.** В некоторые внешние программы может быть включен код, предписывающий иные бизнес-правила поддержки ссылочной целостности по сравнению с теми, которые вы собираетесь реализовать. Из этого следует, что нужно либо еще раз вернуться к вопросу о введении рассматриваемого ограничения внешнего ключа, поскольку в вашей организации не сложилось общее мнение в отношении бизнес-правила, реализуемого с его помощью, либо исправить код самого приложения, чтобы он мог работать с новой версией бизнес-правила (которая является новой с точки зрения этого приложения).
3. **Несуществующий код поддержки ссылочной целостности.** В некоторых внешних программах может даже не быть учтено наличие бизнес-правила поддержки ссылочной целостности, относящегося к рассматриваемым таблицам с данными.

Все внешние программы должны быть обновлены с учетом необходимости обработки любого исключения (исключений), активизируемого базой данных в результате создания нового ограничения внешнего ключа. В следующем коде показано, как внести в прикладной код изменения, необходимые для обработки исключений, активизируемых базой данных:

```
// Код до рефакторинга
stmt = conn.prepare(
  "INSERT INTO Account(AccountID,StatusCode,Balance) "+
  "VALUES (?,?,?)";
stmt.setLong(1,accountId);
stmt.setString(2,statusCode);
stmt.setBigDecimal(3,balance);
stmt.executeUpdate();

// Код после рефакторинга
stmt = conn.prepare(
  "INSERT INTO Account(AccountID,StatusCode,Balance) "+
  "VALUES (?,?,?)";
stmt.setLong(1,accountId);
stmt.setString(2,statusCode);
stmt.setBigDecimal(3,balance);
try {
  stmt.executeUpdate();
} catch (SQLException exception){
      int errorCode = exception.getErrorCode();
      if (errorCode = 2291) {
           handleParentRecordNotFoundError();
      }
      if (errorCode = 2292) {
           handleParentDeletedWhenChildFoundError();
      }
}
```

Операция рефакторинга “Добавление триггера для вычисляемого столбца”

Эта операция предусматривает введение нового триггера, предназначенного для обновления значения, содержащегося в вычисляемом столбце (рис. 8.2). Вычисляемый столбец мог быть ранее введен с помощью операции рефакторинга “Введение вычисляемого столбца” (с. 120).

Обоснование

Необходимость в применении операции “Добавление триггера для вычисляемого столбца” может быть в основном обусловлено тем, что значение, содержащееся в столбце, должно обновляться соответствующим образом при любых изменениях исходных данных. Это действие необходимо осуществлять с помощью базы данных, для того чтобы необходимость его выполнения в приложениях была исключена.

Потенциальные преимущества и недостатки

Если для получения данных, вводимых в вычисляемый столбец, требуются сложные алгоритмы или используются данные, которые находятся в нескольких местах, то триггер должен реализовывать большой объем бизнес-логики. При этом могут возникать несовместимости с аналогичной бизнес-логикой, реализованной в приложениях.

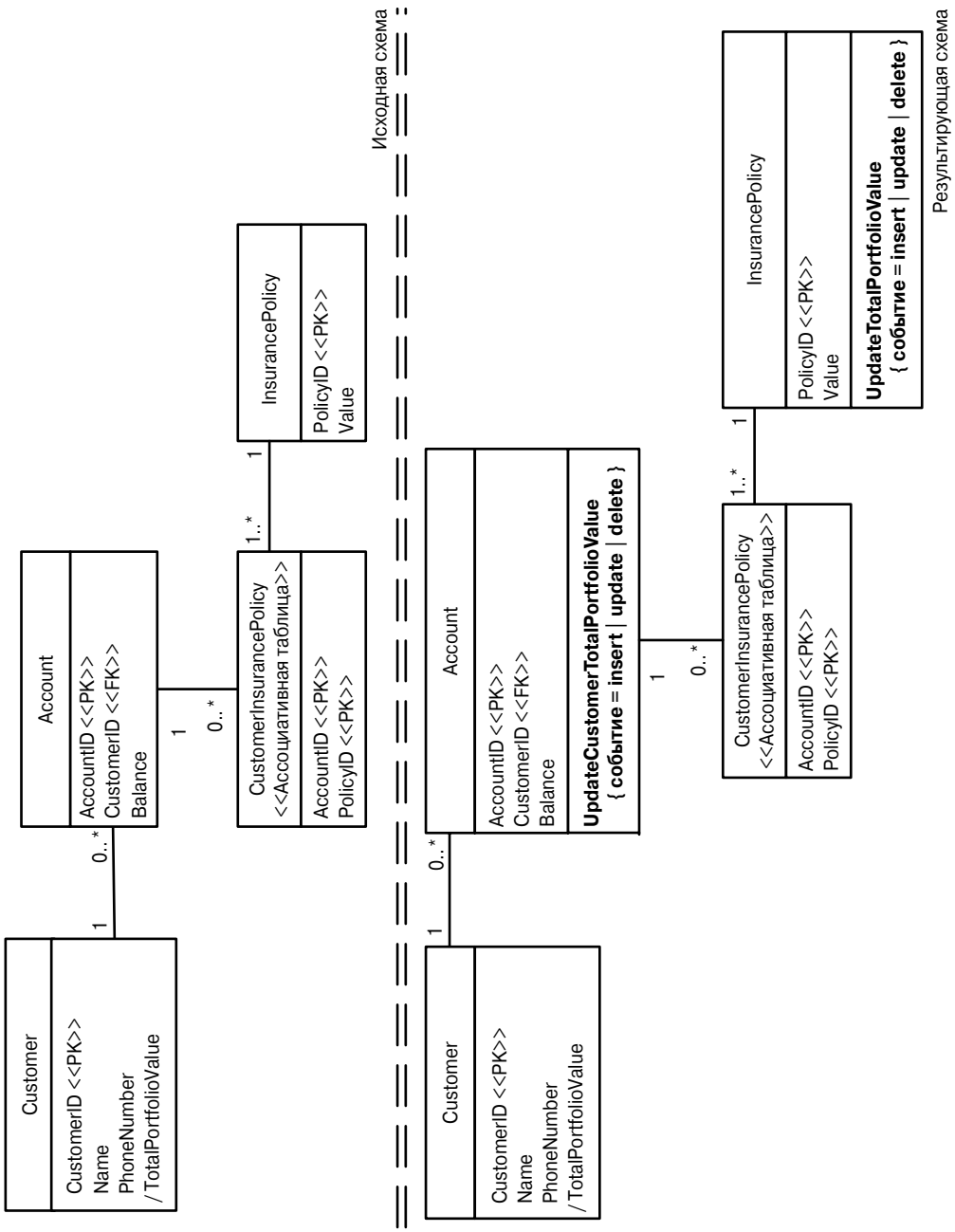


Рис. 8.2. Добавление триггера для вычисления значения `Customer.TotalPortfolioValue`

Может оказаться так, что исходные данные, используемые триггером, обновляются в рамках определенной транзакции. Если операция, выполняемая триггером, оканчивается неудачей, то неудачно завершается и транзакция, вызывая ее откат. Для внешних программ подобные ситуации, скорее всего, будут выглядеть как нежелательный побочный эффект.

Если данные в вычисляемом столбце основаны на данных, находящихся в той же таблице, то может оказаться так, что триггер нельзя будет использовать для обновления значений в вычисляемом столбце, поскольку этого не допускают многие программные продукты баз данных.

Процедура обновления схемы

Применение операции "Добавление триггера для вычисляемого столбца" может оказаться затруднительным в связи с наличием зависимостей в данных, относящихся к вычисляемому столбцу. Это означает, что необходимо выполнить описанные ниже действия.

1. **Определить, могут ли триггеры использоваться для обновления данных в вычисляемом столбце.** Столбец `TotalPortfolioValue` (рис. 8.2) является вычисляемым, поскольку перед его именем стоит косая черта (/). Если столбец `TotalPortfolioValue` и столбцы с исходными данными находятся в одной и той же таблице, то, по-видимому, нельзя будет использовать триггеры для обновления значений данных в столбце `TotalPortfolioValue`.
2. **Определить исходные данные.** Для определения значения вычисляемого столбца требуется выявить исходные данные и узнать, как они должны использоваться.
3. **Узнать, в какой таблице должен находиться вычисляемый столбец.** Необходимо определить, в какой таблице должен находиться вычисляемый столбец, если он еще не существует. Для этого следует ответить на вопрос о том, какую деловую сущность этот вычисляемый столбец описывает лучше всего. Например, индикатор кредитного риска клиента является самым применимым к сущности `Customer`.
4. **Добавить столбец.** Если столбец еще не существует, введите его с помощью команды `ALTER TABLE ADD COLUMN`. Для этого используется операция "Введение нового столбца" (с. 321).
5. **Добавить триггер (триггеры).** Необходимо добавить триггер к каждой таблице, содержащей исходные данные, которые используются для вычисления рассматриваемого значения. В этом случае исходные данные для столбца `TotalPortfolioValue` существуют в таблицах `InsurancePolicy` и `Account`. Поэтому требуются триггеры для каждой из этих таблиц, `UpdateCustomerTotalPortfolioValue` и `UpdateTotalPortfolioValue` соответственно.

В следующем коде показано, как добавить эти два триггера:

```
-- Обновление значения TotalPortfolioValue с помощью триггера
CREATE OR REPLACE TRIGGER
UpdateCustomerTotalPortfolioValue
  AFTER UPDATE OR INSERT OR DELETE
  ON Account
```

```
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
    BEGIN
      UpdateCustomerWithPortfolioValue;
    END;
  END;
/

CREATE OR REPLACE TRIGGER
UpdateCustomerTotalPortfolioValue
  AFTER UPDATE OR INSERT OR DELETE
  ON InsurancePolicy
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECALRE
    BEGIN
      UpdateCustomerWithPortfolioValue;
    END;
  END;
/
```

Процедура переноса данных

Как таковые, данные, подлежащие переносу, отсутствуют, хотя должно быть обеспечено заполнение столбца `Customer.TotalPortfolioValue` с применением значений, полученных путем вычислений. Как правило, такая операция выполняется один раз, в пакетном задании, с помощью одного или нескольких сценариев. В рассматриваемом примере необходимо обновить столбец `Customer.TotalPortfolioValue` в объеме, соответствующем всем существующим строкам в таблице `Customer`, введя в него значения суммы `Account.Balance` и `Policy.Value`, относящиеся к каждому клиенту:

```
UPDATE Customer SET TotalPortfolioValue =
  (SELECT SUM(Account.Balance) + SUM(Policy.Balance)
    FROM Account, CustomerInsurancePolicy, InsurancePolicy
    WHERE
    Account.AccountID = CustomerInsurancePolicy.AccountID
    AND CustomerInsurancePolicy.PolicyID=Policy.PolicyID
    AND Account.CustomerID=Customer.CustomerID
  );
```

Процедура обновления программ доступа

Необходимо выявить все те места во внешних приложениях, где в текущее время выполняются рассматриваемые вычисления, а затем переопределить этот код, чтобы он работал со значениями из столбца `TotalPortfolioValue`; для этого обычно код процедуры вычисления удаляется и заменяется кодом чтения из базы данных. Может оказаться так, что эти вычисления выполняются по-разному в различных приложениях, либо из-за программной ошибки, либо в связи с другим стечением обстоятельств, поэтому потребуется согласовать правильный алгоритм вычисления с предприятием.

Операция рефакторинга “Уничтожение ограничения внешнего ключа”

Эта операция позволяет удалить ограничение внешнего ключа из существующей таблицы, чтобы ее связь с другой таблицей больше не поддерживалась принудительно базой данных (рис. 8.3).

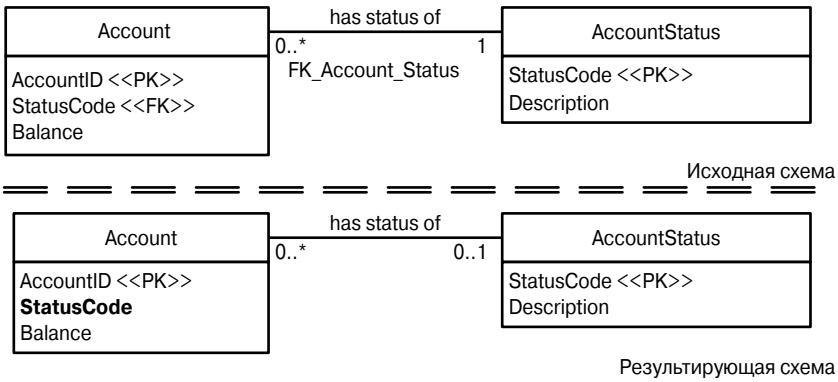


Рис. 8.3. Удаление ограничения внешнего ключа из таблицы `Account`

Обоснование

Основная причина применения операции “Уничтожение ограничения внешнего ключа” состоит в том, что отпадает необходимость поддерживать в дальнейшем зависимости между данными на уровне базы данных. Вместо этого задача обеспечения целостности данных возлагается на внешние приложения. Это особенно важно, если издержки, связанные с поддержкой ограничений ссылочной целостности в базе данных и выражающиеся в снижении производительности, становятся неприемлемыми, и (или) если в разных приложениях применяются различные правила ссылочной целостности.

Потенциальные преимущества и недостатки

Оценка основных преимуществ и недостатков, связанных с применением этой операции рефакторинга, должна быть основана на определении компромисса между производительностью и качеством. Дело в том, что ограничения внешнего ключа гарантируют допустимость данных на уровне базы данных, но за счет того, что проверка соблюдения этих ограничений производится при каждом обновлении исходных данных. А после применения операции “Уничтожение ограничения внешнего ключа” приложения подвергаются риску ввода в базу данных недопустимых данных, если в этих приложениях не выполняется проверка допустимости данных перед их записью в базу данных.

Процедура обновления схемы

Для удаления ограничения внешнего ключа необходимо применить либо команду `ALTER TABLE DROP CONSTRAINT`, либо `ALTER TABLE DISABLE CONSTRAINT`. Преимущество последнего подхода состоит в том, что он позволяет обнаружить в базе данных

существование связи, даже притом, что само ограничение уже не предписывается. В следующем коде показаны два способа, с помощью которых можно удалить ограничение внешнего ключа между столбцами `Account.StatusCode` и `AccountStatus.StatusCode`, приведенными на рис. 8.3. Первый способ предусматривает только удаление ограничения, а во втором способе осуществляется его отмена, и поэтому в базе данных по-прежнему можно обнаружить, что когда-то была необходимость в его использовании. Авторы рекомендуют применять второй подход:

```
ALTER TABLE Account DROP CONSTRAINT FK_Account_Status;
ALTER TABLE Account DISABLE CONSTRAINT FK_Account_Status;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных необходимость в переносе данных отсутствует.

Процедура обновления программ доступа

Необходимо выявить, а затем исправить все внешние программы, предназначенные для внесения изменений в столбец (столбцы) данных, на котором было определено ограничение внешнего ключа. Во внешних программах требуется найти проявления следующих двух проблем. Во-первых, необходимо обновить отдельные приложения, чтобы по-прежнему соблюдались соответствующие правила ссылочной целостности. Эти правила могут изменяться, но, вообще говоря, необходимо ввести в каждое приложение программные средства, обеспечивающие существование в таблице `AccountStatus` определенной строки при любой такой ситуации, когда на эту строку формируется ссылка в таблице `Account`. Из этого следует, что необходимо либо определить наличие такой строки (в частности, если таблица, в данном случае `AccountStatus`, невелика, то ее можно кэшировать в памяти), либо, в случае необходимости, добавить новую строку в таблицу. Во-вторых, следует обеспечить обработку исключительных ситуаций. После удаления рассматриваемого ограничения внешнего ключа база данных больше не будет активизировать относящиеся к нему исключения ссылочной целостности, поэтому в код всех внешних программ должны быть введены соответствующие изменения.

Операция рефакторинга “Введение каскадного удаления”

Эта операция обеспечивает автоматическое удаление в базе данных соответствующих “дочерних” записей после удаления “родительской” записи (рис. 8.4).

Следует отметить, что вместо удаления “дочерних” записей достаточно удалить ссылку на “родительскую” запись в “дочерних” записях. Этот вариант может использоваться, только если столбец (столбцы) внешнего ключа в “дочерних” таблицах допускает применение NULL-значений, хотя его реализация может привести к появлению большого количества зависших строк.

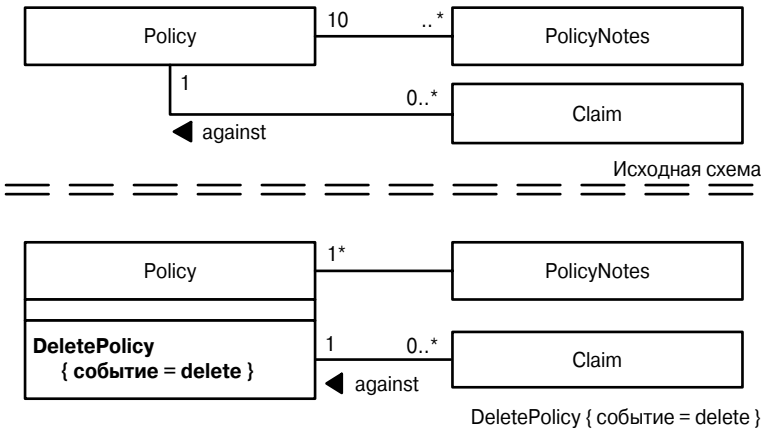


Рис. 8.4. Ввод в действие средств каскадного удаления на столбце `Policy`

Обоснование

Необходимость в применении операции "Введение каскадного удаления" может быть обусловлена тем, что требуется обеспечить ссылочную целостность в данных путем удаления должным образом связанных строк после удаления относящейся к ним строки "родительской" таблицы.

Потенциальные преимущества и недостатки

При выполнении этой операции рефакторинга следует учитывать описанные ниже потенциальные преимущества и недостатки.

- **Взаимоблокировка.** При реализации каскадных удалений необходимо избегать циклических зависимостей, поскольку в противном случае могут возникать взаимоблокировки. Современные базы данных обнаруживают взаимоблокировки и выполняют откат транзакции, которая вызвала взаимоблокировку.
- **Случайное массовое удаление.** При использовании этой операции рефакторинга необходимо соблюдать исключительную осторожность. Например, теоретически возможно, что после удаления одной строки из таблицы `CorporateDivision` будут также удалены все соответствующие ей строки из таблицы `Employee`. Но может оказаться так, что в результате неосторожного удаления строки, представляющей крупное подразделение, будут удалены тысячи строк с данными о сотрудниках, которые нельзя удалять.
- **Дублирующиеся функциональные средства.** Такие инфраструктуры обеспечения доступа к базе данных, как Hibernate (`www.hibernate.org`) и Toplink корпорации Oracle (`www.oracle.com`), позволяют автоматизировать управление связями между объектами, поэтому при применении рассматриваемой операции рефакторинга могут обнаружиться дублирующиеся функциональные средства.

Процедура обновления схемы

Чтобы иметь возможность использовать операцию “Введение каскадного удаления”, необходимо выполнить описанные ниже действия.

1. **Выявить данные, подлежащие удалению.** Необходимо определить, какие “дочерние” строки должны быть удалены после удаления “родительской” строки. Например, при удалении заказа должны быть также удалены все компоненты, связанные с ним. Такое действие является рекурсивным; “дочерние” строки могут, в свою очередь, иметь “дочерние” строки, которые также нужно удалить, а это означает, что придется применить операцию “Введение каскадного удаления” и к этим строкам. Авторы не рекомендуют проводить рассматриваемую операцию рефакторинга исчерпывающим образом; вместо этого следует применить эту операцию рефакторинга только к одному набору таблиц, полностью реализовать предусмотренные в ней действия, выполнить проверку и только после этого переходить к следующему набору таблиц. Незначительные изменения проще реализовать, чем большие.
2. **Выбрать механизм каскадного удаления.** Операции каскадного удаления могут быть реализованы с помощью триггеров или ограничений ссылочной целостности, которые определены опцией `DELETE CASCADE`. (Однако не все поставщики баз данных поддерживают эту опцию.)
3. **Реализовать каскадное удаление.** При использовании первого подхода необходимо разработать триггер, который удаляет все строки, “дочерние” по отношению к “родительской” строке, после удаления последней. Этот вариант в наибольшей степени применим, если требуется обеспечить детализированный контроль над тем, какие строки должны быть удалены после удаления “родительской” строки. Недостатком этого подхода является то, что должен быть написан код реализации соответствующего функционального средства. Кроме того, реализация этого подхода без достаточного учета зависимостей между многочисленными триггерами, одновременно вызываемыми на выполнение, может привести к возникновению взаимоблокировок. При использовании второго подхода должны быть определены ограничения ссылочной целостности путем ввода в действие опции `DELETE CASCADE` команды `ALTER TABLE MODIFY CONSTRAINT` языка SQL. Если будет выбран этот подход, то необходимо определить в базе данных ограничения ссылочной целостности, но эта задача может оказаться очень трудоемкой применительно к той базе данных, для которой еще не приходилось определять ограничения ссылочной целостности (поскольку потребуется использовать операцию рефакторинга “Добавление ограничения внешнего ключа”, см. с. 244, почти ко всем связям в базе данных). Основное преимущество этого подхода состоит в том, что не приходится писать какой-либо новый код, поскольку база данных автоматически удаляет “дочерние” строки после удаления “родительской” строки. Недостаток последнего подхода в том, что при его использовании усложняется задача отладки.

На рис. 8.4 показано, как реализовать операцию "Введение каскадного удаления" применительно к таблице Policy, используя подход, основанный на применении триггера. Триггер DeletePolicy, код которого приведен ниже, удаляет все строки из таблицы PolicyNotes или Claim, которые связаны с удаляемой строкой в таблице Policy.

```
-- Создание триггера для удаления строк PolicyNotes и Claim
CREATE OR REPLACE TRIGGER DeletePolicy
AFTER DELETE ON Policy
FOR EACH ROW
DECLARE
BEGIN
  DeletePolicyNotes();
  DeletePolicyClaims();
END;
/
```

В следующем коде показано, как реализовать операцию "Введение каскадного удаления", используя ограничения ссылочной целостности с опцией DELETE CASCADE:

```
ALTER TABLE POLICYNOTES ADD CONSTRAINT FK_DELETEPOLICYNOTES
FOREIGN KEY (POLICYID)
REFERENCES POLICY (POLICYID)
ON DELETE CASCADE
ENABLE
;

ALTER TABLE CLAIMS ADD CONSTRAINT FK_DELETEPOLICYCLAIM
FOREIGN KEY (POLICYID)
REFERENCES POLICY (POLICYID)
ON DELETE CASCADE
ENABLE
;
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных необходимость в переносе данных отсутствует.

Процедура обновления программ доступа

В связи с применением этой операции рефакторинга необходимо удалить весь прикладной код, который в настоящее время реализует функциональные средства удаления "дочерних" строк. При этом может возникнуть одна проблема, состоящая в том, что в одних приложениях реализовано удаление, а в других — нет. Предположим, что в одном приложении после удаления соответствующей строки из таблицы Order удаляются строки из таблицы OrderItem, а в другом приложении это не предусмотрено. Реализация каскадного удаления в такой базе данных может оказать непредвиденное влияние на работу второго приложения, возможно, в лучшую сторону. Но общий вывод состоит в том, что необходимо соблюдать исключительную осторожность; не следует предполагать, что во всех приложениях реализованы одни и те же правила поддержки ссылочной целостности, несмотря на то "очевидное" требование, что так и должно быть.

Может также возникнуть необходимость обработать все новые типы сообщений об ошибках, возвращаемых базой данных в случае неудачного завершения операции каскадного удаления. В следующем коде показано, как изменяется прикладной код, применяемый до и после осуществления операции “Введение каскадного удаления”:

```
// Код до рефакторинга
private void deletePolicy (Policy policyToDelete) {
  Iterator policyNotes =
  policyToDelete.getPolicyNotes().iterator();
  for (Iterator iterator = policyNotes; iterator.hasNext();) {
    PolicyNote policyNote = (PolicyNote) iterator.next();
    DB.remove(policyNote);
}
  DB.remove(policyToDelete);
}

// Код после рефакторинга
private void deletePolicy (Policy policyToDelete) {
DB.remove(policyToDelete);
}
```

Если используются какие-либо инструментальные средства отображения O-R (“Object-Relation” — “объект-отношение”), то необходимо внести изменения в файл отображения, чтобы можно было задать опцию каскадного удаления в отображении:

```
// Отображение после рефакторинга
<hibernate-mapping>
  <class name="Policy" table="POLICY">
  ......
  <one-to-many name="policyNotes"
               class="PolicyNotes"
               cascade="all-delete-orphan"
  />
  ......
  </class>
</hibernate-mapping>
```

Операция рефакторинга “Введение физического удаления”

Эта операция позволяет удалить существующий столбец, который указывает, что строка удалена (такой способ удаления называется *программным* или *логическим удалением*), и вместо этого действительно удалить строку из таблицы (например, выполнить физическое удаление) (рис. 8.5). Эта операция рефакторинга противоположна операции “Введение программного удаления” (с. 246).

Обоснование

Необходимость в применении операции “Введение физического удаления” может быть в основном обусловлена потребностью в сокращении размера таблиц, что приводит к повышению производительности и упрощению запросов, поскольку в них больше не нужно будет проверять, отмечена ли какая-либо строка как удаленная.

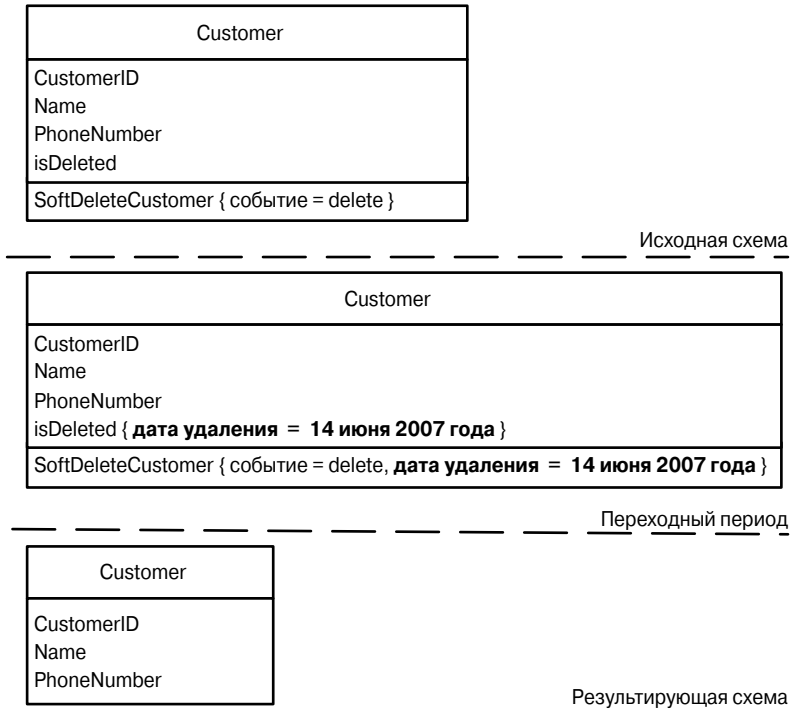


Рис. 8.5. Применение операции физического удаления к таблице `Customer`

Потенциальные преимущества и недостатки

Единственным недостатком этой операции рефакторинга является потеря данных, которые когда-то были актуальными, а теперь представляют интерес только для истории, хотя и может использоваться операция "Введение триггера для накопления исторических данных" (с. 250), если это потребуется.

Процедура обновления схемы

Как показано на рис. 8.5, при осуществлении операции "Введение физического удаления" необходимо в первую очередь удалить столбец идентификации, поскольку он больше не потребуется; в рассматриваемом случае таким столбцом является `Customer.isDeleted`; см. описание операции рефакторинга "Удаление столбца" (с. 112). Затем следует удалить весь код, обычно код триггера (но иногда и код приложения), который обновляет индикаторный столбец, в данном случае столбец `Customer.isDeleted`. Может быть также предусмотрен код, который устанавливает начальные значения, равные `FALSE`, если применяется булев индикатор, или задает предопределенную дату, если используется отметка даты/времени. Такие программные средства обычно реализуются с помощью ограничения столбца, определяющего заданное по умолчанию значение, а это ограничение должно быть автоматически удалено наряду со столбцом. Может также быть определен код триггера, который задает значение, равное `TRUE`, или вводит текущее значение даты/времени при по-

пытке удаления строки. По всей вероятности, потребуется лишь удалить этот триггер. В следующем коде показано, как удалить столбец `Customer.isDeleted`:

```
DROP COLUMN Customer ALTER TABLE isDeleted;
```

Процедура переноса данных

Необходимо удалить все строки с данными из таблицы `Customer`, для которых значение `isDeleted` установлено равным `TRUE`, поскольку они представляют собой строки, которые были логически удалены. Но прежде чем удалить эти строки, необходимо обновить или, возможно, удалить все данные, которые ссылаются на логически удаленные данные. Как правило, такая операция выполняется один раз, в пакетном задании, с помощью одного или нескольких сценариев. Следует также рассмотреть возможность архивирования всех строк, отмеченных как предназначенные для удаления, чтобы иметь возможность отменить рассматриваемую операцию рефакторинга, если это потребуется. В следующем коде показано, как удалить из таблицы `Customer` строки, для которых флажок `Customer.isDeleted` установлен равным `TRUE`:

```
-- Удалить данные о клиентах, предназначенные для удаления
DELETE FROM Customer WHERE isDeleted = TRUE;
```

Процедура обновления программ доступа

При использовании операции рефакторинга “Введение физического удаления” необходимо внести корректировки во внешние программы, получающие доступ к данным, проводя эту работу в двух направлениях. Во-первых, необходимо исключить из операторов `SELECT` ссылки на столбец `Customer.isDeleted`. Во-вторых, следует обновить весь логический код удаления:

```
// Код до рефакторинга
public void customerDelete(Long customerIdToDelete)
throws Exception {
  PreparedStatement stmt = null;
  try {
    stmt =  DB.prepare("UPDATE Customer "+
              "SET isDeleted = ? "+
              "WHERE CustomerID =  ?");
    stmt.setLong(1, Boolean.TRUE);
    stmt.setLong(2, customerIdToDelete);
    stmt.execute();
  } catch (SQLException SQLexc){
    DB.HandleDBException(SQLexc);
  }
  finally {DB.cleanup(stmt);}
}

// Код после рефакторинга
public void customerDelete(Long customerIdToDelete) throws Exception {
  PreparedStatement stmt = null;
  try {
    stmt =  DB.prepare("DELETE FROM Customer "+
```

```
                        "WHERE CustomerID =  ?");
    stmt.setLong(1, customerIdToDelete);
    stmt.execute();
  } catch (SQLException SQLexc){
    DB.HandleDBException(SQLexc);
  }
  finally {DB.cleanup(stmt);}
}
```

Операция рефакторинга "Введение программного удаления"

Эта операция позволяет ввести в существующую таблицу столбец с флажком, который указывает, что строка была удалена (такое удаление называется *программным*, или *логическим*), вместо действительного удаления строки (физического удаления) (рис. 8.6). Эта операция рефакторинга является противоположной по отношению к операции "Введение физического удаления" (с. 243).

Обоснование

Необходимость в применении операции "Введение программного удаления" в основном обусловлена требованием сохранять все данные приложения, как правило, для использования их в целях изучения истории изменений.

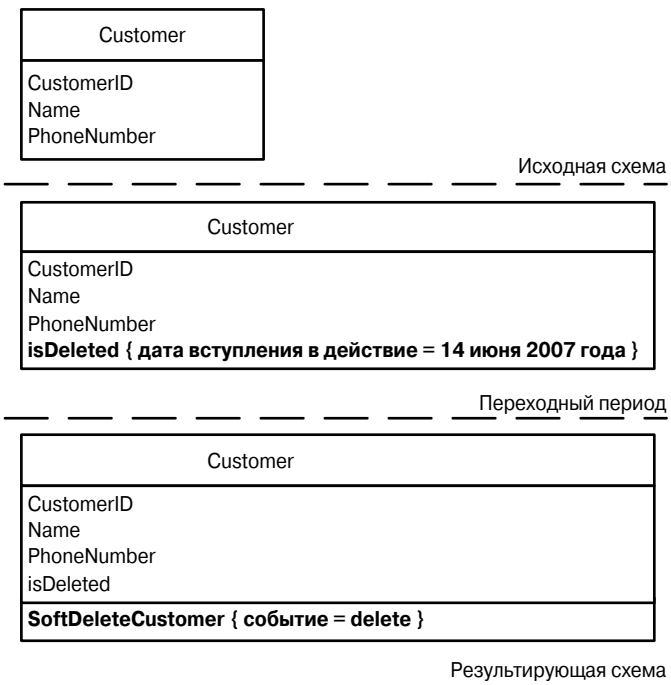


Рис. 8.6. Применение операции программного удаления к таблице `Customer`

Потенциальные преимущества и недостатки

Осуществление рассматриваемой операции рефакторинга потенциально способно оказать отрицательное влияние на производительность по двум описанным ниже причинам. Во-первых, в базе данных приходится хранить все строки, которые были отмечены как удаленные. Это может привести к существенному увеличению объема используемого дискового пространства и снижению производительности запросов. Во-вторых, в приложениях с этого времени приходится выполнять дополнительную работу, проводя различие между удаленными и неудаленными строками, что также влечет за собой снижение производительности и вместе с тем может потребовать увеличения сложности кода.

Процедура обновления схемы

Как показывает схема, приведенная на рис. 8.6, для осуществления операции “Введение программного удаления” необходимо выполнить описанные ниже действия.

1. **Ввести столбец идентификации.** Необходимо ввести в таблицу `Customer` новый столбец, который позволяет определить, является ли строка удаленной или нет; см. описание преобразования “Введение нового столбца” (с. 321). В этом столбце обычно содержатся либо булевы поля, в которых находится значение `TRUE`, если строка удалена, и `FALSE` — в противном случае, или отметка даты/времени, указывающая, когда строка была удалена. В рассматриваемом примере вводится булев столбец `isDeleted`, который не должен допускать использование NULL-значений. (См. описание операции рефакторинга “Преобразование столбца в недопускающий NULL-значения” на с. 216.)
2. **Предусмотреть способ обновления флажков.** Обновление столбца `Customer.isDeleted` может осуществляться либо в приложении (приложениях), либо непосредственно в базе данных с использованием триггеров. Авторы предпочитают подход, основанный на использовании триггеров, поскольку он проще и позволяет избежать того, что в приложениях обновление столбца не будет выполняться должным образом.
3. **Разработать код удаления.** Должен быть написан и проверен код, обеспечивающий обновление столбца — индикатора удаления соответствующим образом после “псевдоудаления” строки. В случае применения булева столбца устанавливается значение, равное `TRUE`, а в случае использования отметки даты/времени в качестве значения столбца задаются текущая дата и время.
4. **Разработать код вставки.** После выполнения любой операции вставки необходимо установить должным образом значение столбца, выполняющего функции индикатора удаления; при этом в случае использования булева столбца задается значение `FALSE`, а при использовании отметки даты/времени вводится предопределенная дата (например, 1 января 5000 года). Такую доработку можно легко реализовать с использованием операции рефакторинга “Введение заданного по умолчанию значения” (с. 213) или с помощью триггера.

В следующем коде показано, как ввести дополнительный столбец `Customer.isDeleted` и присвоить ему по умолчанию значение `FALSE`:

```
ALTER TABLE Customer ADD isDeleted BOOLEAN;
ALTER TABLE Customer MODIFY isDeleted DEFAULT FALSE;
```

А в приведенном ниже коде показано, как создать триггер, который вызывается вместо команды `DELETE` языка SQL и присваивает флажку `Customer.isDeleted` значение `TRUE`. В этом коде строка с данными перед удалением копируется, обновляется индикатор удаления, а затем строка вставляется назад в таблицу после удаления ее оригинала.

```
-- Создание массива для хранения удаленных данных Customers
CREATE OR REPLACE PACKAGE SoftDeleteCustomerPKG
AS
  TYPE ARRAY IS TABLE OF Customer%ROWTYPE INDEX BY BINARY_INTEGER;
  oldvals  ARRAY;
  empty    ARRAY;
END;
/

-- Инициализация массива
CREATE OR REPLACE TRIGGER SoftDeleteCustomerBefore
BEFORE DELETE ON Customer
BEGIN
  SoftDeleteCustomerPKG.oldvals := SoftDeleteCustomerPKG.empty;
END;
/

-- Перехват удаляемых строк
CREATE OR REPLACE TRIGGER SoftDeleteCustomerStore
BEFORE DELETE ON Customer
FOR EACH ROW
DECLARE
  i NUMBER DEFAULT SoftDeleteCustomerPKG.oldvals.COUNT+1;
BEGIN
  SoftDeleteCustomerPKG.oldvals(i).CustomerID := :old.CustomerID;
  deleteCustomer.oldvals(i).Name := :old.Name;
  deleteCustomer.oldvals(i).PhoneNumber := :old.PhoneNumber;
END;
/

-- Обратная вставка данных о клиентах с флажком isDeleted,
-- установленным равным TRUE
CREATE OR REPLACE TRIGGER SoftDeleteCustomerAdd
AFTER DELETE ON Customer
DECLARE
BEGIN
  FOR i IN 1 .. SoftDeleteCustomerPKG.oldvals.COUNT LOOP
    insert into Customer(CustomerID,Name,PhoneNumber,isDeleted)
    values( deleteCustomer.oldvals(i).CustomerID,
      deleteCustomer.oldvals(i).Name,
      deleteCustomer.oldvals(i).PhoneNumber,
      TRUE);
```

```
  END LOOP;
END;
/
```

Процедура переноса данных

Как таковые, данные, подлежащие удалению, отсутствуют, но в ходе реализации этой операции рефакторинга во всех строках столбцу `Customer.isDeleted` должно быть присвоено соответствующее значение, заданное по умолчанию. Как правило, такая операция выполняется один раз, в пакетном задании, с помощью одного или нескольких сценариев.

Процедура обновления программ доступа

При использовании операции рефакторинга “Введение программного удаления” необходимо откорректировать все внешние программы, получающие доступ к данным. Во-первых, теперь запросы на чтение должны быть откорректированы так, чтобы в них обеспечивалось чтение из базы данных только таких данных, которые не отмечены как удаленные. В приложениях следует добавить конструкцию `WHERE` ко всем запросам `SELECT`, например, такую, как `WHERE isDeleted = FALSE`. Вместо корректировки всех запросов на чтение можно использовать операцию рефакторинга “Инкапсуляция таблицы в представление” (с. 266), с тем чтобы для возврата из таблицы `Customer` строк, в которых `isDeleted` не равно `TRUE`, служило представление. Еще один вариант состоит в том, что может быть применена операция рефакторинга “Добавление метода чтения” (с. 263), для того чтобы соответствующая конструкция `WHERE` была реализована только в одном месте. Во-вторых, необходимо откорректировать методы удаления. Во всех внешних программах необходимо заменить операции физического удаления операциями обновления, которые обновляют значение в столбце `Customer.isDeleted` вместо физического удаления строки. Например, оператор `DELETE FROM Customer WHERE PKColumn = nnn` должен быть заменем оператором `UPDATE Customer SET isDeleted = TRUE WHERE PKColumn = nnn`. Еще один вариант состоит в том, что, как было указано выше, можно ввести триггер удаления, который предотвращает удаление и обновляет значение `Customer.isDeleted`, присваивая значение `TRUE`.

В следующем коде показано, как можно задать начальное значение столбца `Customer.isDeleted`:

```
UPDATE Customer SET
  isDeleted = FALSE WHERE isDeleted IS NULL;
```

А в приведенном ниже коде демонстрируется метод чтения для простого объекта `Customer`, применяемый до и после проведения операции рефакторинга “Введение программного удаления”.

```
// Код до рефакторинга
stmt.prepare(
"SELECT CustomerId, Name, PhoneNumber " +
"FROM Customer" +
"WHERE " +
```

```
"  CustomerId = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
"SELECT CustomerId, Name, PhoneNumber " +
"FROM Customer" +
"WHERE " +
"  CustomerId = ? "+
"  AND isDeleted = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.setBoolean(2,false);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Фрагменты кода, обозначенные как применяемые до и после проведения операции рефакторинга "Введение программного удаления", показывают, каким образом изменяется метод удаления в результате осуществления этой операции:

```
// Код до рефакторинга
stmt.prepare("DELETE FROM Customer WHERE CustomerID=?");
stmt.setLong(1, customer.getCustomerID);
stmt.executeUpdate();

// Код после рефакторинга
stmt.prepare("UPDATE Customer SET isDeleted = ?"+
             "WHERE CustomerID=?");
stmt.setBoolean(1, true);
stmt.setLong(2, customer.getCustomerID);
stmt.executeUpdate();
```

Операция рефакторинга "Введение триггера для накопления исторических данных"

Эта операция позволяет ввести новый триггер, предназначенный для накопления информации об изменениях в данных в целях изучения истории внесения изменения или проведения аудита (рис. 8.7).

Обоснование

Необходимость в применении операции "Введение триггера для накопления исторических данных" в основном обусловлена требованием передать функции отслеживания изменений в данных самой базе данных. Такая стратегия гарантирует, что в случае модификации важных исходных данных в любом приложении это изменение можно будет отследить или подвергнуть аудиту.

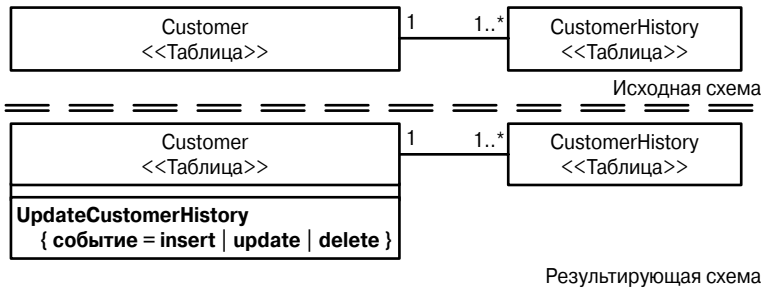


Рис. 8.7. Ввод в действие триггера, который предназначен для накопления данных об истории внесения изменений в таблицу `Customer`

Потенциальные преимущества и недостатки

Основные факторы, которые должны рассматриваться перед применением этой операции рефакторинга, касаются производительности, поскольку ввод в действие триггера приводит к увеличению времени, расходуемого на обновление строк, например, в таблице `Customer` (рис. 8.7).

Кроме того, может возникнуть необходимость откорректировать приложения и предусмотреть в них передачу контекстной информации о пользователе, чтобы в базе данных можно было зарегистрировать сведения о том, кто внес то или иное изменение.

Процедура обновления схемы

Для осуществления операции “Введение триггера для накопления исторических данных” необходимо выполнить описанные ниже действия.

1. **Определить действия, применительно к которым должны накапливаться исторические сведения.** Допустим, что в таблице `Customer` разрешается вставлять, обновлять и удалять данные, но необходимость в отслеживании всех изменений отсутствует. Например, предположим, что интерес представляет лишь отслеживание операций обновления и удаления, но не первоначально выполненных операций вставки.
2. **Определить столбцы, для которых необходимо накапливать исторические сведения.** Должны быть определены те столбцы в таблице `Customer`, по отношению к которым необходимо обеспечить отслеживание изменений. Например, предположим, что требуется отслеживать только изменения в столбце `PhoneNumber`, а другие столбцы интереса не представляют.
3. **Определить способ регистрации исторических данных.** При этом могут применяться два основных варианта: во-первых, можно предусмотреть применение универсальной таблицы, которая отслеживает все изменения исторических данных в базе данных, а во-вторых, ввести соответствующую таблицу хронологии применительно к каждой таблице, для которой необходимо регистрировать исторические данные. Подход, основанный на использовании одной таблицы, недостаточно хорошо масштабируется, но является приемлемым для небольших баз данных.

4. **Ввести таблицу с историческими данными.** Если таблица с историческими данными еще не существует, то ее необходимо создать с помощью команды `CREATE TABLE`.
5. **Ввести в действие триггер (триггеры).** Необходимо ввести в действие соответствующий триггер (триггеры) с помощью команды `CREATE OR REPLACE TRIGGER`. Этой цели можно достичь, предусмотрев использование триггера, который копирует исходный образ строки и вставляет его в таблицу `CustomerHistory`. Еще один вариант состоит в том, чтобы сравнивать значения в контролируемых столбцах до и после внесения изменений и сохранять описания этих изменений, скажем, в таблице `CustomerHistory`.

В следующем коде показано, как создать таблицу `CustomerHistory` и обновить ее с помощью триггера `UpdateCustomerHistory`. В данном случае производится сбор информации об изменениях в каждом столбце таблицы `Customer` и запись этой информации в таблицу `CustomerHistory`:

```
-- Создание таблицы CustomerHistory
CREATE TABLE CustomerHistory
(
  CustomerID NUMBER,
  OldName VARCHAR2(32),
  NewName VARCHAR2(32),
  ChangedBy NUMBER,
  ChangedOn DATE
);

-- Триггер аудита для таблицы Customer
CREATE OR REPLACE TRIGGER
UpdateCustomerHistory
AFTER INSERT OR UPDATE OR DELETE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
-- Обработка операций обновления строк
  IF UPDATING THEN
    IF NOT (NVL(:OLD.Name = :NEW.Name,FALSE) OR
    (:OLD.Name IS NULL AND :NEW.Name IS NULL)) THEN
     CreateCustomerHistoryRow( :NEW.CustomerId, :NEW.Name, :OLD.Name,
User);
    END IF;
  END IF;

-- Обработка операций удаления строк
  IF DELETING THEN
    IF (:OLD.Name IS NOT NULL) THEN
      CreateCustomerHistoryRow (:OLD.CustomerId,:OLD.Name,NULL,User);
    END IF;
  END IF;
```

```
  IF INSERTING THEN
    IF (:NEW.Name IS NOT NULL) THEN
      CreateCustomerHistoryRow(:NEW.CustomerId,:NEW.Name,NULL,User);
    END IF;
  END IF;
END;
/
```

Процедура переноса данных

При осуществлении этой операции рефакторинга базы данных перенос данных обычно не требуется. Но если есть необходимость отслеживать операции вставки, то можно создать строку хронологии для каждой существующей строки таблицы `Customer`. Такой подход является наиболее применимым, если таблица `Customer` включает столбец, в котором регистрируется исходная дата создания; в противном случае может потребоваться вырабатывать значение даты вставки. (Для этого проще всего использовать текущую дату.)

Процедура обновления программ доступа

При осуществлении операции “Введение триггера для накопления исторических данных” необходимо выполнить описанные ниже действия.

1. **Отменить формирование исторических данных в приложениях.** После того как будут введены в действие триггеры, предназначенные для сбора исторической информации, необходимо определить наличие во внешних приложениях всех таких существующих программных средств, в которых применяется код создания исторической информации, а затем откорректировать эти приложения.
2. **Обновить код презентации.** Если в настоящее время применяются внешние программы, которые служат для отображения исторических сведений, сформированных в приложениях, то эти программы необходимо откорректировать в целях использования информации, содержащейся, например, в таблице `CustomerHistory`.
3. **Передать пользовательский контекст в базу данных.** Если есть необходимость предусмотреть регистрацию с помощью триггера базы данных информации о том, какой пользователь внес изменения в данные, то требуется получить пользовательский контекст. В некоторых приложениях, особенно в тех, которые созданы с применением Web-технологий или многоуровневых технологий, передача такой информации может быть не предусмотрена, поэтому следует обновить эти приложения с учетом указанного условия. Еще один вариант состоит в том, что можно предусмотреть применение заданного по умолчанию пользовательского контекста, если он не предоставляется.

Кроме того, может потребоваться соответствующим образом преобразовать схему и откорректировать проект таблицы, чтобы добавить столбцы, предназначенные для регистрации данных о том, кто и когда внес изменения в соответствующую строку, хотя такая доработка может быть не предусмотрена в самой операции рефакторинга. Наиболее широко применимый подход состоит в том, что в основные таблицы вводятся такие столб-

цы, как `UserCreated`, `CreationDate`, `UserLastModified` и `LastModifiedDate` (см. описание операции "Введение нового столбца" на с. 342). В двух столбцах с данными о пользователях могут содержаться идентификаторы пользователей, применимые в качестве внешних ключей для таблицы с данными о пользователях. Может также потребоваться предусмотреть использование дополнительной таблицы с данными о пользователях (см. описание операции "Введение поисковой таблицы" на с. 196).

Глава 9

Операции рефакторинга архитектуры

Операции рефакторинга архитектуры представляют собой изменения, проведение которых в целом способствует улучшению характера взаимодействия внешних программ с базой данных. Операции рефакторинга архитектуры перечислены ниже.

- Операция рефакторинга “Добавление методов CRUD”.
- Операция рефакторинга “Добавление зеркальной таблицы”.
- Операция рефакторинга “Добавление метода чтения”.
- Операция рефакторинга “Инкапсуляция таблицы в представление”.
- Операция рефакторинга “Введение вычислительного метода”.
- Операция рефакторинга “Введение индекса”.
- Операция рефакторинга “Введение таблицы только для чтения”.
- Операция рефакторинга “Перенос метода из базы данных”.
- Операция рефакторинга “Перенос метода в базу данных”.
- Операция рефакторинга “Замена метода (методов) представлением”.
- Операция рефакторинга “Замена представления методом (методами)”.
- Операция рефакторинга “Использование официально заданного источника данных”.

Операция рефакторинга “Добавление методов CRUD”

Эта операция предусматривает ввод в действие четырех хранимых процедур (методов), обеспечивающих создание, выборку, обновление и удаление (Creation, Retrieval, Update, Deletion — CRUD) данных, представляющих деловую сущность (рис. 9.1). После ввода таких хранимых процедур в действие доступ к исходным таблицам ограничивается применением только этих процедур и программ, находящихся в распоряжении администраторов базы данных.

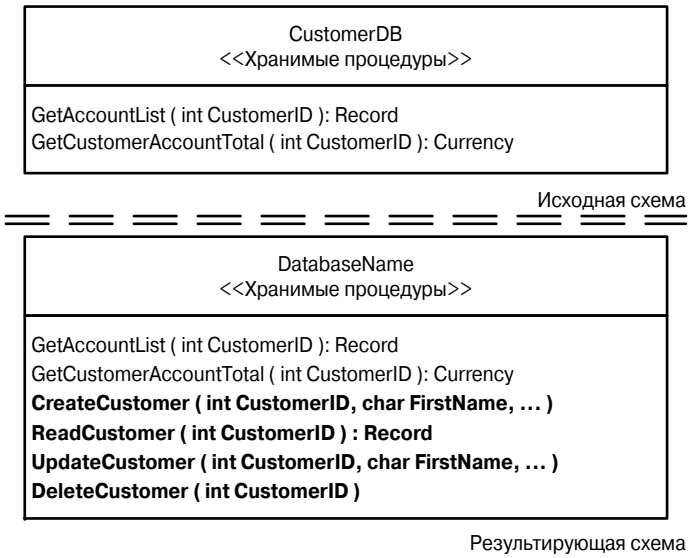


Рис. 9.1. Добавление методов CRUD к таблице `Customer`

Обоснование

Необходимость в применении операции рефакторинга "Добавление методов CRUD" обусловлена перечисленными ниже причинами.

- **Инкапсуляция средств доступа к данным.** Хранимые процедуры представляют собой распространенный способ инкапсуляции средств доступа к базе данных, хотя не столь эффективный, как инфраструктуры доступа к базе данных (см. главу 1 "Эволюционная разработка базы данных").
- **Отделение прикладного кода от определений таблиц базы данных.** Хранимые процедуры представляют собой эффективный способ отделения кода приложений от таблиц базы данных. Они позволяют вносить изменения в таблицы базы данных без корректировки кода приложений.
- **Реализация средств управления защитой доступа (Security Access Control — SAC) на основе сущности.** При использовании таких средств в приложениях вместо непосредственного вызова исходных таблиц происходит вызов соответствующих хранимых процедур. Хранимые процедуры CRUD предоставляют возможность реализовать один из подходов на основе сущностей к созданию средств SAC. В программных продуктах баз данных обычно предусмотрена возможность использовать средства управления защитой доступа к данным на уровне таблицы, столбца, а иногда и на уровне строки. Но если данные, относящиеся к таким сложным деловым сущностям, как `Customer`, хранятся в нескольких таблицах, то в случае применения исключительно лишь средств управления защитой доступа, ориентированных на данные, такие средства, реализованные на уровне сущности, могут стать чрезвычайно сложными. К счастью, в большинстве программных

продуктов баз данных реализованы также средства управления защитой доступа с помощью методов, поэтому обеспечивается возможность реализовать средства SAC на основе сущности с помощью методов CRUD.

Потенциальные преимущества и недостатки

Основным преимуществом инкапсуляции средств доступа к базе данных указанным способом является то, что благодаря этому упрощается применение операций рефакторинга к схеме таблицы. При реализации таких распространенных операций рефакторинга схемы, как “Переименование таблицы” (с. 148), “Перемещение столбца” (с. 139) и “Разбиение столбца” (с. 172), необходимо лишь подвергнуть рефакторингу соответствующие методы CRUD, которые получают доступ к соответствующим объектам базы данных (предполагается, что отсутствуют какие-либо иные внешние программы, непосредственно обращающиеся к схеме таблицы).

К сожалению, этот подход к инкапсуляции базы данных требует определенных затрат. Методы (хранимые процедуры, хранимые функции и триггеры) отражают специфику продуктов, выпускаемых определенными поставщиками баз данных, поэтому, например, методы из СУБД Oracle нелегко перенести в СУБД Sybase. Более того, нет никакой гарантии, что методы, работающие с одной версией базы данных, удастся легко перенести на более новую версию, поэтому в связи с проведением рассматриваемой операции рефакторинга может также возрасти нагрузка, обусловленная необходимостью обновления программного обеспечения. Еще одним недостатком является почти полное отсутствие гибкости. Рассмотрим, что произойдет, если потребуется получить доступ только к некоторой части деловой сущности. При этом приходится принимать решение о том, следует ли каждый раз извлекать из базы данных все данные рассматриваемой деловой сущности или, возможно, ввести в действие методы CRUD для работы с требуемой частью сущности. Не менее важная проблема возникает, если требуются данные, которые относятся к разным сущностям, возможно, для формирования отчета. При этом приходится принимать решение о том, следует ли вызывать хранимые процедуры для чтения данных каждой необходимой деловой сущности, а затем выборки искомых данных, или применять операцию рефакторинга “Добавление метода чтения” (с. 263) для выборки именно тех данных, которые необходимы для формирования отчета.

Процедура обновления схемы

Для обновления схемы базы данных необходимо выполнить описанные ниже действия.

1. **Определить необходимую деловую информацию.** Прежде всего следует определить деловую сущность, такую как сущность `Customer` (рис. 9.1), для которой должны быть реализованы методы CRUD. После того как станет известно, применительно к какой сущности должны быть инкапсулированы средства доступа, необходимо определить, какие исходные данные в базе данных относятся к этой сущности.
2. **Написать хранимые процедуры.** Необходимо написать по крайней мере четыре хранимых процедуры, во-первых, для создания сущности, во-вторых, для чтения данных сущности на основе ее первичного ключа, в-третьих, для обновления сущности и, в-четвертых, для ее удаления. С помощью операции рефакторинга базы данных “Добавление метода чтения” (с. 263) могут быть дополнительно

введены хранимые процедуры для выборки данных сущности с использованием средств, отличных от первичного ключа.

3. **Проверить хранимые процедуры.** Одним из лучших способов выполнения такой проверки является использование подхода к разработке, основанного на тестировании (Test-Driven Development — TDD), как описано в главе 1 "Эволюционная разработка базы данных".

На рис. 9.1 показано, как ввести в действие методы CRUD применительно к сущности `Customer`. Ниже приведен код хранимой процедуры `ReadCustomer`. Код других хранимых процедур не требует пояснений.

```
CREATE OR REPLACE PACKAGE CustomerCRUD AS
    TYPE customerType IS REF CURSOR RETURN Customer%ROWTYPE;
    PROCEDURE ReadCustomer
      (readCustomerId IN NUMBER,customers OUT customerType);
    PROCEDURE CreateCustomer(....);
    PROCEDURE UpdateCustomer(....);
    PROCEDURE DeleteCustomer(....);
    END CustomerCRUD;
    /

    CREATE OR REPLACE PACKAGE BODY CustomerCRUD AS

    PROCEDURE ReadCustomer
      (readCustomerId IN NUMBER,customerReturn OUT customerType) IS
    BEGIN
    OPEN refCustomer FOR
      SELECT * FROM Customer WHERE CustomerID = readCustomerId;
    END ReadCustomer;

    END CustomerCRUD;
/
```

Определение соглашений об именовании

В целях повышения удобства для чтения в методах CRUD следует соблюдать общепринятые соглашения об именовании. Как показано на рис. 9.1, авторы предпочитают использовать формат наподобие `CreateCustomer`, `ReadCustomer`, `UpdateCustomer` и `DeleteCustomer`, хотя имеет смысл применять также формат `CustomerCreate`, `CustomerRead`, `CustomerUpdate` и `CustomerDelete`. Так или иначе, необходимо выбрать один подход и придерживаться его.

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

При вводе в действие методов CRUD необходимо выявить все те места во внешних приложениях, где в текущее время используется рассматриваемая сущность, а затем, по мере необходимости, подвергнуть рефакторингу такие участки кода внешних приложений, для того чтобы в них применялись новые хранимые процедуры. Может также оказаться, что в отдельных программах требуются различные подмножества данных, иначе говоря, для некоторых программ требуется немного больше данных по сравнению с другими. Кроме того, может обнаружиться, что операцию рефакторинга “Добавление методов CRUD” необходимо применить к нескольким сущностям одновременно, что потенциально становится весьма значительным изменением.

В первом примере кода показано, что класс Java сущности `Customer` первоначально применялся для передачи кода SQL, реализованного с применением программных конструкций, в базу данных для выборки соответствующих данных. А второй пример кода показывает, как подвергнуть этот класс Java рефакторингу, для того чтобы в нем осуществлялся вызов хранимой процедуры:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT FirstName, Surname, PhoneNumber FROM Customer " +
    "WHERE CustomerId=?");
stmt.setLong(1,customerId);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt = conn.prepareCall("begin ? := ReadCustomer(?); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.setLong(2, customerId);
stmt.execute();
ResultSet rs = stmt.getObject(1);
```

Операция рефакторинга “Добавление зеркальной таблицы”

Эта операция позволяет создать зеркальную таблицу, т.е. точный дубликат существующей таблицы в одной базе данных, в другой базе данных (рис. 9.2).

Обоснование

Необходимость в применении операции “Добавление зеркальной таблицы” может быть обусловлена несколькими причинами, описанными ниже.

- **Повышение производительности запросов.** Выполнение запросов применительно к какому-то конкретному множеству таблиц может происходить медленно из-за того, что база данных находится в удаленном месте, поэтому применение заранее заполненной таблицы в базе данных, развернутой на локальном сервере, может способствовать повышению общей производительности.

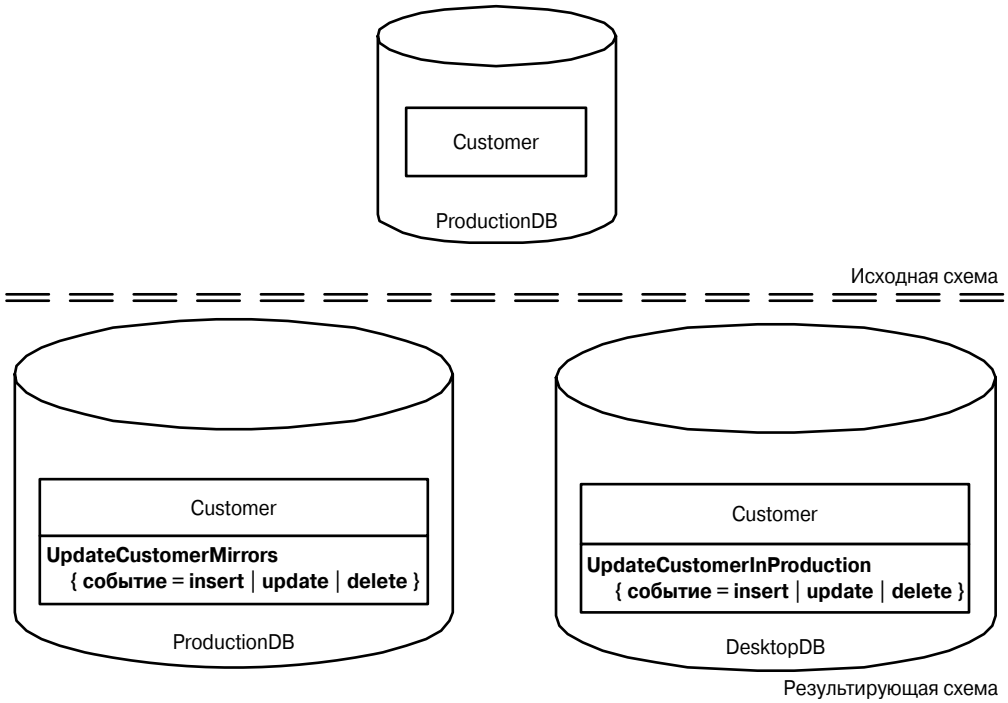


Рис. 9.2. Добавление зеркальной таблицы `Customer`

- **Создание избыточных данных.** Для многих приложений требуется получение данных с помощью запросов из других баз данных в режиме реального времени. Если такие данные содержатся в таблице (таблицах) локальной базы данных, то зависимость приложения от удаленной базы (баз) данных уменьшается, поскольку появляется буферное пространство, с которым могут работать приложения, если доступ к удаленной базе данных прекращается или происходит ее останов для обслуживания.
- **Замена избыточных операций чтения.** Часто наблюдается такая ситуация, что в нескольких внешних программах (или, применительно к рассматриваемой теме, в хранимых процедурах) используются одни и те же запросы выборки данных. В таком случае запросы к таблице удаленной базы данных могут быть заменены запросами к зеркальной таблице в локальной базе данных, которая представляет собой реплику таблицы из удаленной базы данных.

Потенциальные преимущества и недостатки

Основной проблемой, о которой следует помнить, вводя зеркальную таблицу, является возникновение так называемых *устаревших данных*. Устаревшие данные возникают, если одна из зеркальных таблиц обновляется, а другая — нет; следует помнить, что потенциально может обновляться либо исходная таблица, либо зеркальная. Эта проблема становится все более острой по мере увеличения количества зеркальных

таблиц, создаваемых в других базах данных, в рассматриваемом случае применительно к таблице `Customer`. Как показано на рис. 9.2, необходимо ввести в действие своего рода стратегию синхронизации.

Процедура обновления схемы

Как показано на рис. 9.2, для обновления схемы базы данных при осуществлении операции рефакторинга “Добавление зеркальной таблицы” необходимо выполнить описанные ниже действия.

1. **Определить местоположение.** Необходимо принять решение о том, где должна находиться зеркальная таблица, т.е. таблица `Customer`; в рассматриваемом случае для зеркального отображения этой таблицы применяется база данных `DesktopDB`.
2. **Ввести в действие зеркальную таблицу.** Таблица `DesktopDB.Customer` создается в другой базе данных с использованием команды `CREATE TABLE` языка SQL.
3. **Определить стратегию синхронизации.** Для синхронизации данных должен быть принят подход, обеспечивающий обновление в реальном времени, подобный показанному на рис. 9.2, если конечные пользователи требуют, чтобы информация всегда была актуальной.
4. **Предусмотреть выполнение операций обновления.** Если требуется обеспечить применение операций обновления к таблице `DesktopDB.Customer`, то необходимо предусмотреть способ синхронизации данных в направлении от таблицы `DesktopDB.Customer` к таблице `Customer`. Если таблица `DesktopDB.Customer` является обновляемой, то такой способ зеркального отображения принято называть *одноранговым зеркальным отображением*; он известен также под названием зеркального отображения по принципу “ведущий/ведомый”.

Ниже приведен пример операторов DDL, которые должны быть применены к таблице `DesktopDB.Customer`.

```
CREATE TABLE Customer (
  CustomerID  NUMBER NOT NULL,
  Name  VARCHAR(40),
  PhoneNumber VARCHAR2(40),
  CONSTRAINT PKCustomer
    PRIMARY KEY (CustomerID)
);

COMMENT ON Customer 'Mirror table of Customer on Remote Location'
```

Процедура переноса данных

Вначале необходимо скопировать все соответствующие исходные данные в зеркальную таблицу, а затем ввести в действие применяемую стратегию синхронизации (обновление в реальном времени или использование средств репликации базы данных). Как описано ниже, предусмотрено несколько стратегий реализации, которые можно применить к принятой стратегии синхронизации.

1. **Периодическое обновление.** Использование планируемого задания, которое синхронизирует таблицы `Customer` и `DesktopDB.Customer`. Это задание должно позволять учитывать изменения данных в обеих таблицах и обновлять данные в обоих направлениях. Подход, основанный на периодических обновлениях, обычно является наиболее приемлемым при эксплуатации приложений, основанных на технологии хранилищ данных.
2. **Репликация баз данных.** В программных продуктах баз данных предусмотрены средства, позволяющие осуществлять репликацию таблиц в обоих направлениях; такой способ синхронизации данных называется репликацией с несколькими ведущими. При этом база данных поддерживает обе таблицы в синхронизированном состоянии. Вообще говоря, если должна быть предусмотрена возможность обновления пользователями обеих таблиц, и `Customer`, и `DesktopDB.Customer`, то следует использовать именно этот подход. Если применяемый программный продукт баз данных предоставляет такую возможность, то рекомендуется использовать именно эти средства, а не реализовывать решение, основанное на коде собственной разработки.
3. **Использование средств синхронизации на основе триггера.** На таблице `Customer` должны быть созданы триггеры, позволяющие распространить изменения в исходных данных на таблицу `DesktopDB.Customer`, а на таблице `DesktopDB.Customer` — триггеры, распространяющие изменения в этой таблице на таблицу `Customer`. Этот подход позволяет применять определяемый пользователем код для синхронизации данных, что может потребоваться, если синхронизации подлежат сложные объекты данных, но при этом должны быть разработаны все необходимые триггеры, а эта задача может оказаться трудоемкой.

В следующем коде показано, как синхронизировать данные в таблицах `ProductionDB.Customer` и `DesktopDB.Customer` с использованием триггеров:

```
CREATE OR REPLACE TRIGGER
  UpdateCustomerMirror
  AFTER UPDATE OR INSERT OR DELETE
  ON Customer
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
    DECLARE
      BEGIN
        IF DELETING THEN
          DeleteCustomerMirror;
        END IF;
        IF (UPDATING OR INSERTING) THEN
          UpdateCustomerMirror;
        END IF;
      END;
  END;
  /

CREATE OR REPLACE TRIGGER
  UpdateCustomer
  AFTER UPDATE OR INSERT OR DELETE
```

```
  ON CustomerMirror
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF DELETING THEN
      DeleteCustomer;
    END IF;
    IF (UPDATING OR INSERTING) THEN
      UpdateCustomer;
    END IF;
  END;
END;
/
```

Процедура обновления программ доступа

Процедура обновления программ доступа осуществляется с учетом причин, по которым была введена в действие зеркальная таблица, в данном случае `DesktopDB.Customer`. Например, если создается зеркальная копия таблицы `Customer` из удаленной базы данных в локальной базе данных для упрощения доступа, то необходимо, чтобы приложение подключалось к локальной базе данных, а не к удаленной базе данных `ProductionDB`, после того как возникает необходимость выполнить запрос к таблице `Customer`. Могут быть также заданы альтернативные свойства соединения, которые позволяют выбирать для использования при выполнении запросов к данным базу данных `DesktopDB` или `ProductionDB`, как показано в следующем коде:

```
// Код до рефакторинга
stmt = remoteDB.prepare("select CustomerID,Name,PhoneNumber FROM
Customer WHERE CustomerID = ?");
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt = localDB.prepare("select CustomerID,Name,PhoneNumber FROM
Customer WHERE CustomerID = ?");
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга “Добавление метода чтения”

Эта операция позволяет ввести в действие метод, в данном случае хранимую процедуру, для реализации операции выборки данных, представляющих нуль или больше деловых сущностей, из базы данных (рис. 9.3).

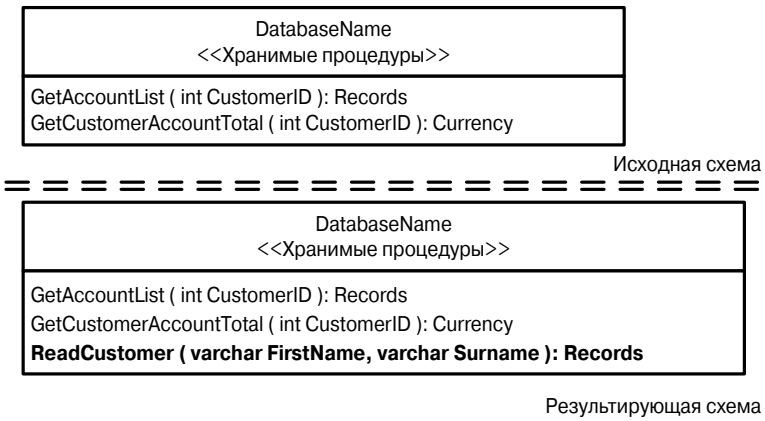


Рис. 9.3. Добавление метода чтения для таблицы `Customer`

Обоснование

Необходимость в применении операции "Добавление метода чтения" может быть обусловлена тем, что должна быть обеспечена инкапсуляция программных средств, применяемых для выборки из базы данных конкретных данных в соответствии с определенными критериями. Такой метод чтения, часто реализуемый с помощью хранимой процедуры, может потребоваться для замены одного или нескольких операторов `SELECT`, реализованных в коде приложения и/или в коде формирования отчетов. Еще одна часто возникающая причина может быть связана с необходимостью реализовать согласованную стратегию поиска для рассматриваемой деловой сущности.

Иногда эта операция рефакторинга применяется для поддержки операции рефакторинга "Введение программного удаления" (с. 246) или "Введение физического удаления" (с. 243). После применения одной из последних указанных операций рефакторинга изменяется способ удаления данных; в первом случае строка отмечается как удаленная, а во втором происходит физическое удаление данных. После подобного изменения способа удаления может потребоваться реализация значительного количества изменений в различных приложениях для обеспечения того, чтобы операции выборки выполнялись правильно. Если средства выборки данных инкапсулированы в хранимой процедуре, после чего предусмотрен вызов должным образом этой хранимой процедуры, то реализация двух указанных операций рефакторинга, связанных с удалением данных, намного упрощается, поскольку необходимые программные средства выборки должны быть переопределены только в одном месте (в хранимой процедуре).

Потенциальные преимущества и недостатки

Основное преимущество инкапсуляции программных средств выборки данных в описанной форме, часто в сочетании с операцией "Добавление методов CRUD" (с. 255), состоит в том, что ее применение способствует упрощению рефакторинга схемы таблицы. К сожалению, этот подход к инкапсуляции базы данных требует определенных затрат. Код хранимых процедур отражает специфические особенности конкретных

продуктов, предоставляемых поставщиками баз данных, поэтому возможности переноса этого кода на другие платформы становятся ограниченными, а если разработка хранимых процедур выполнена недостаточно качественно, то их применение может привести к уменьшению общей производительности базы данных.

Процедура обновления схемы

Для обновления схемы базы данных необходимо выполнить описанные ниже действия.

1. **Определить необходимые данные.** Должны быть определены данные, подлежащие выборке, которые могут находиться в нескольких разных таблицах.
2. **Определить критерии.** Требуется также определить способ описания подмножества данных, подлежащих выборке. Например, может потребоваться обеспечить выборку информации, относящейся к банковским счетам, остаток на которых превышает указанную сумму, или которые были открыты в указанном отделении банка, или на которых происходило движение денежных средств в течение определенного промежутка времени, или же применить сочетание подобных критериев. Следует отметить, что в состав подобных критериев может входить или не входить первичный ключ.
3. **Написать и проверить хранимую процедуру.** Авторы — убежденные сторонники написания полного, 100%-го регрессионного тестового набора для хранимых процедур. А в качестве лучшего подхода авторы рекомендуют подход на основе тестирования (Test-Driven Development — TDD), в соответствии с которым вначале следует обеспечить тестирование и лишь затем приступать к написанию кода хранимых процедур [**9**; **10**].

На рис. 9.3 показано, как ввести в действие хранимую процедуру чтения для сущности `Customer`, которая принимает в качестве параметров имя и фамилию клиента. Код хранимой процедуры `ReadCustomer` приведен ниже. Она написана исходя из того, что ей передаются такие параметры, как `S%` и `Ambler`, после чего происходит возврат данных о всех клиентах с фамилией `Ambler`, имя которых начинается с буквы `S`.

```
PROCEDURE ReadCustomer
  (
  firstNameToFind IN VARCHAR,
  lastNameToFind IN VARCHAR,
  customerRecords OUT CustomerREF
) IS
BEGIN
OPEN refCustomer FOR
  SELECT * FROM Customer WHERE
  Customer.FirstName = firstNameToFind
  AND Customer.LastName = lastNameToFind;
END ReadCustomer;
/
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

После ввода в действие хранимой процедуры чтения необходимо выявить все те участки кода во внешних приложениях, где происходит чтение данных, а затем подвергнуть этот код рефакторингу, для того чтобы в нем должным образом вызывалась новая хранимая процедура. Может быть также обнаружено, что для отдельных программ требуются различные подмножества данных, в связи с чем, вероятно, потребуется применить операцию "Добавление метода чтения" несколько раз, для каждой из основных коллекций элементов данных.

В следующем коде показано, что в приложениях в базу данных передаются операторы `SELECT` для выборки информации о клиентах, а после выполнения операции рефакторинга происходит только вызов хранимой процедуры:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT * FROM Customer " +
    "WHERE Customer.FirstName=? AND Customer.LastName=?");
stmt.setString(1,firstNameToFind);
stmt.setString(2,lastNameToFind);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt = conn.prepareCall("begin ? := ReadCustomer(?,?,?); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.setString(2, firstNameToFind);
stmt.setString(3,lastNameToFind);
stmt.execute();
ResultSet rs = stmt.getObject(1);
while (rs.next()) {
  getCustomerInformation(rs);
}
```

Операция рефакторинга "Инкапсуляция таблицы в представление"

Эта операция позволяет заключить средства доступа к существующей таблице в представление (рис. 9.4).

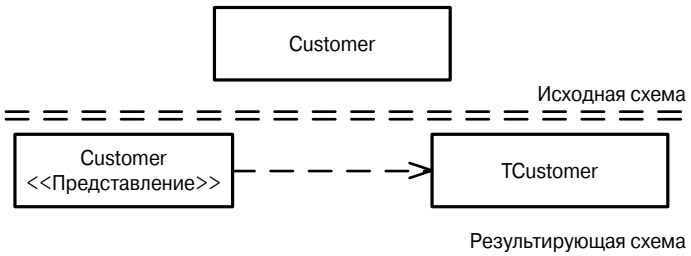


Рис. 9.4. Инкапсуляция таблицы `TCustomer` в представление

Обоснование

Необходимость в применении операции “Инкапсуляция таблицы в представление” может быть обусловлена двумя причинами. Во-первых, может потребоваться оградить существующую таблицу от приложений, поскольку намечено в дальнейшем подвергнуть ее рефакторингу. Если в приложениях предусмотрен доступ к исходным данным с помощью представлений, а не непосредственный доступ с помощью таблиц, то уменьшается непосредственная связь приложений с этой таблицей, что позволяет проще обеспечить рефакторинг этой таблицы. Например, предположим, что требуется инкапсулировать средства доступа к таблице с помощью представления, прежде чем применить такие операции рефакторинга, как “Переименование столбца” (с. 145) или “Удаление столбца” (с. 112). Во-вторых, может быть предусмотрен ввод в действие средств управления защитой доступа (Security Access Control — SAC) применительно к базе данных. Это может быть осуществлено путем инкапсуляции доступа к таблице с помощью некоторых представлений и последующего ограничения доступа к таблице и различным представлениям соответствующим образом. После этого некоторые пользователи, если таковые вообще имеются, смогут получать непосредственный доступ к исходной таблице, а основной массе пользователей будут вместо этого предоставляться права доступа к одному или нескольким представлениям. Первым шагом этого процесса является инкапсуляция доступа к исходной таблице с помощью представления, а затем введение новых представлений по мере необходимости.

Потенциальные преимущества и недостатки

Как показано на рис. 9.4, такая операция рефакторинга осуществима лишь при том условии, что применяемая база данных поддерживает такой же уровень доступа с помощью представлений, как и с помощью таблиц. Например, если предусмотрена возможность обновлять таблицу `Customer` с помощью внешних программ, то применяемая база данных должна поддерживать обновляемые представления. В противном случае вместо этого необходимо предусмотреть применение операции рефакторинга “Добавление методов CRUD” (с. 255), которая позволяет инкапсулировать средства доступа к этой таблице.

Процедура обновления схемы

Как показано на рис. 9.4, эта операция рефакторинга является несложной. На первом этапе необходимо переименовать существующую таблицу, в рассматриваемом случае — присвоить ей новое имя, `TCustomer`. На втором этапе вводится представление с именем исходной таблицы, т.е. `Customer`. С точки зрения внешних программ, получающих доступ к таблице, в действительности ничего не изменяется. Просто эти программы получают доступ к исходным данным с помощью представления, но фактически при этом не обнаруживают каких-либо отличий от исходной таблицы.

Один из способов переименования таблицы `Customer` состоит в использовании конструкции `RENAME TO` команды `ALTER TABLE` языка SQL, как показано ниже. Если применяемая база данных не поддерживает эту конструкцию, то необходимо создать новую таблицу `TCustomer`, скопировать в нее данные из таблицы `Customer`, после чего удалить таблицу `Customer`. Пример реализации такого подхода приведен в описании операции рефакторинга "Переименование таблицы" (с. 148). Независимо от того, каким образом была переименована таблица, на следующем этапе необходимо ввести в действие представление с помощью команды `CREATE VIEW`:

```
ALTER TABLE Customer RENAME TO TCustomer;

CREATE VIEW Customer AS
  SELECT * FROM TCustomer;
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

Обновлять какие-либо программы доступа нет необходимости, поскольку представление идентично по своей структуре исходной таблице.

Операция рефакторинга "Введение вычислительного метода"

Эта операция позволяет ввести в действие новый метод, как правило, хранимую функцию, которая реализует определенные вычисления с использованием данных, хранимых в базе данных (рис. 9.5).

Обоснование

Необходимость в применении операции "Введение вычислительного метода" может быть обусловлена несколькими описанными ниже причинами.

- **Повышение производительности приложения.** Общую производительность приложения можно повысить путем реализации вычислений, для которых требуется значительный объем данных, непосредственно в базе данных и последующей выдачи только готового ответа. Это позволяет избежать необходимости передавать по сети данные, которые требуются для проведения вычислений.

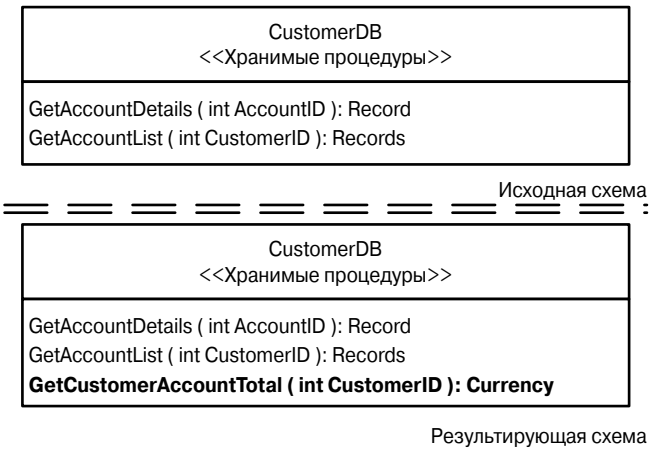


Рис. 9.5. Ввод в действие метода вычисления для таблицы `Customer`

- **Реализация общих, широко применяемых алгоритмов вычисления.** Любые алгоритмы вычисления могут быть реализованы с помощью нескольких существующих методов, поэтому имеет смысл объединить все необходимые средства вычисления в одну хранимую процедуру, которая вызывается по мере необходимости.
- **Выполнение операции “Введение вычисляемого столбца” (с. 120).** Программные средства, необходимые для вычисления значения столбца, можно реализовать с помощью хранимой функции.
- **Замена вычисляемого столбца.** Может быть решено заменить вычисляемый столбец хранимой процедурой, которая вызывается для проведения необходимых вычислений. После этого для удаления столбца следует применить операцию рефакторинга “Удаление столбца” (с. 112).

Потенциальные преимущества и недостатки

Если количество программных средств, реализованных в базе данных, слишком велико, это может стать причиной возникновения узкого места производительности, если база данных не рассчитана на подобную нагрузку. В связи с этим может потребоваться либо перейти к использованию масштабируемой базы данных, либо ограничить количество реализованных в ней функциональных средств.

Процедура обновления схемы

Для соответствующего обновления схемы базы данных необходимо написать, а затем проверить применяемую хранимую процедуру. Если программные средства, применяемые для выполнения соответствующих вычислений, уже реализованы в существующих хранимых процедурах, то последние должны быть подвергнуты рефакторингу в целях вызова новой хранимой процедуры, выполняющей вычисления. В следующем коде показано, как ввести хранимую функцию `GetCustomerAccountTotal` в базу данных `CustomerDB`:

```
CREATE OR REPLACE FUNCTION getCustomerAccountTotal
(CustomerID IN NUMBER) RETURN NUMBER IS
customerTotal NUMBER;
  BEGIN
    SELECT SUM(Amount) INTO customerTotal FROM Policy
      WHERE PolicyCustomerId=CustomerID;
        RETURN customerTotal;
      EXCEPTION WHEN no_data_found THEN
        RETURN 0;
  END;
END;
/
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

При вводе в действие метода вычисления необходимо выявить во внешних приложениях все участки кода, в которых реализованы вычисления, а затем подвергнуть этот код рефакторингу, для того чтобы в нем соответствующим образом вызывалась новая хранимая процедура. В следующем коде показано, что до проведения рефакторинга вычисления выполнялись с помощью языка Java, а после этого вызывалась соответствующая хранимя функция:

```
// Код до рефакторинга
private BigDecimal getCustomerTotalBalance() {
BigDecimal customerTotalBalance = new BigDecimal(0);
for (Iterator iterator = customer.getPolocies().iterator();
iterator.hasNext();) {
  Policy policy = (Policy) iterator.next();
  customerTotalBalance.add(policy.getBalance());
}
return customerTotalBalance;
}

// Код после рефакторинга
private BigDecimal getCustomerTotalBalance() {
  BigDecimal customerTotalBalance = new BigDecimal(0);
  stmt = connection.prepareCall("{call getCustomerAccountTotal(?)}");
  stmt.registerOutParameter(1, Types.NUMBER);
  stmt.setString(1, customer.getId);
  stmt.execute();
  customerTotalBalance = stmt.getBigDecimal(1);
  return customerTotalBalance;
}
```

Может быть обнаружено, что вычисление реализуется в разных приложениях немного различными способами, возможно потому, что рассматриваемое бизнес-правило со временем изменилось, а приложение не было обновлено; может таже существовать какая-то другая действительная причина для этих различий. Так или иначе, необходимо согласовать все проводимые изменения с теми лицами, которые заинтересованы в разработке соответствующего проекта.

Операция рефакторинга “Введение индекса”

Эта операция позволяет ввести новый индекс, либо уникального, либо неуникального типа (рис. 9.6).

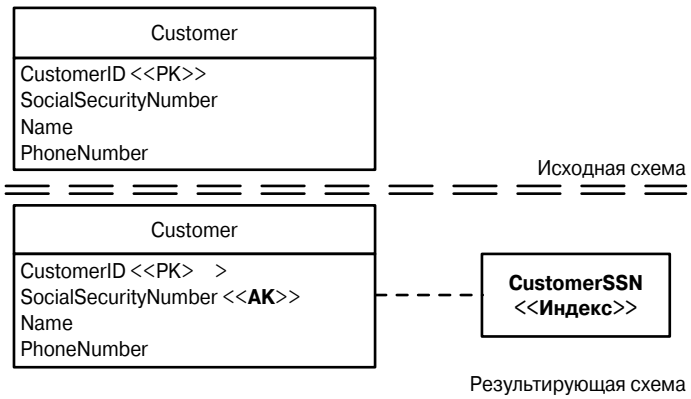


Рис. 9.6. Введение индекса для таблицы `Customer`

Обоснование

Необходимость во введении индекса на таблице может быть обусловлена требованием повысить производительность запросов чтения в базе данных. Введение индекса может также потребоваться для создания первичного ключа для таблицы или поддержки внешнего ключа к другой таблице в связи с проведением операции “Осуществление стратегии консолидированных ключей” (с. 197).

Потенциальные преимущества и недостатки

Если на таблице задано слишком много индексов, это приводит к снижению производительности операций обновления, вставки или удаления, выполняемых применительно к рассматриваемой таблице. Следует отметить, что при введении уникального индекса часто возникает такая ситуация, что эта попытка оканчивается неудачей из-за наличия дублирующихся значений в существующих данных; в связи с этим возникает необходимость удалить такие дубликаты и только после этого переходить к выполнению рассматриваемой операции рефакторинга.

Процедура обновления схемы

Задача осуществления операции "Введение индекса" может оказаться сложной из-за наличия зависимостей в данных, что потенциально потребует обновления и данных, и прикладного кода. Для такого обновления необходимо выполнить описанные ниже действия.

1. **Определить тип индекса.** Следует принять решение о том, требуется ли уникальный или неуникальный индекс, для чего необходимо проанализировать бизнес-правила, относящиеся к атрибутам рассматриваемых данных, и назначение этих данных. Например, в Соединенных Штатах гражданам присваивают уникальные номера социального страхования (Social Security Number — SSN). А в большинстве компаний клиентам присваивают уникальные номера. Но номер телефона может оказаться неуникальным (поскольку одним и тем же номером пользуются несколько человек). Поэтому столбцы `SSN` и `CustomerNumber` могли бы в принципе рассматриваться как применимые для определения уникального индекса, а для столбца `TelephoneNumber`, по-видимому, должен быть определен неуникальный индекс.
2. **Ввести новый индекс.** В примере, приведенном на рис. 9.6, должен быть введен новый индекс для таблицы `Customer` на основе столбца `SocialSecurityNumber` с использованием команды `CREATE INDEX` языка SQL.
3. **Предоставить дополнительное дисковое пространство.** При осуществлении операции рефакторинга "Введение индекса" необходимо запланировать свои действия с учетом увеличения потребности в дисковом пространстве, в связи с чем может потребоваться распределить дополнительное дисковое пространство.

Процедура переноса данных

При выполнении рассматриваемой операции рефакторинга перенос данных как таковой не требуется, но при введении уникального индекса необходимо проверить соответствующий столбец, в данном случае `Customer.SocialSecurityNumber`, на наличие дублирующихся значений. Проблема наличия дублирующихся значений должна быть решена путем обновления дубликатов; если же это невозможно, то должно быть принято решение об использовании неуникального индекса. В следующем коде показано, как определить, имеются ли дубликаты значений в столбце `Customer.SocialSecurityNumber`:

```
-- Поиск всех дублирующихся значений в столбце
Customer.SocialSecurityNumber
SELECT SocialSecurityNumber,COUNT(SocialSecurityNumber)
  FROM Customer
  GROUP BY SocialSecurityNumber
  HAVING COUNT(SocialSecurityNumber)>1;
```

Если будут обнаружены какие-либо дублирующиеся значения, то необходимо внести в них изменения и только после этого переходить к осуществлению операции "Введение индекса". В следующем коде показан один из способов выполнения приведенной выше рекомендации. Прежде всего необходимо воспользоваться значениями столбца

Customer.CustomerID в качестве точки отсчета для поиска дублирующихся строк, а затем заменить дубликаты значений с применением операции рефакторинга “Обновление данных” (с. 329):

```
-- Создание временной таблицы с дубликатами
CREATE TABLE temp_customer
AS
  SELECT * FROM Customer parent
  WHERE CustomerID != (SELECT MAX(CustomerID)
    FROM Customer child
    WHERE parent.SocialSecurityNumber = child.SocialSecurityNumber);

-- Создание уникального индекса
CREATE UNIQUE INDEX Customer_Name_UNQ
  ON Customer(SocialSecurityNumber)
;
```

Процедура обновления программ доступа

Для обновления программ доступа необходимо вначале проанализировать зависимости, чтобы определить, какие внешние программы получают доступ к данным в таблице Customer. Во-первых, эти внешние программы должны обрабатывать исключения базы данных, которые активизируются базой данных в том случае, если в программах вырабатываются дублирующиеся значения. Во-вторых, необходимо внести изменения в запросы, чтобы в них использовался новый индекс в целях повышения производительности выборки данных. Некоторые программные продукты баз данных позволяют указывать, какой индекс должен использоваться при выборке строк из базы данных; для этого служат подсказки, пример применения которых приведен ниже.

```
SELECT /*+ INDEX(Customer_Name_UNQ) */
CustomerID,
SocialSecurityNumber
FROM Customer
WHERE
  SocialSecurityNumber = 'XXX-XXX-XXX';
```

Операция рефакторинга “Введение таблицы только для чтения”

Эта операция позволяет создать хранилище данных, предназначенное только для чтения, на основе существующей таблицы базы данных (рис. 9.7). Предусмотрены два способа реализации этой операции рефакторинга: с помощью таблицы, заполняемой в режиме реального времени, или таблицы, заполняемой в пакетном задании.

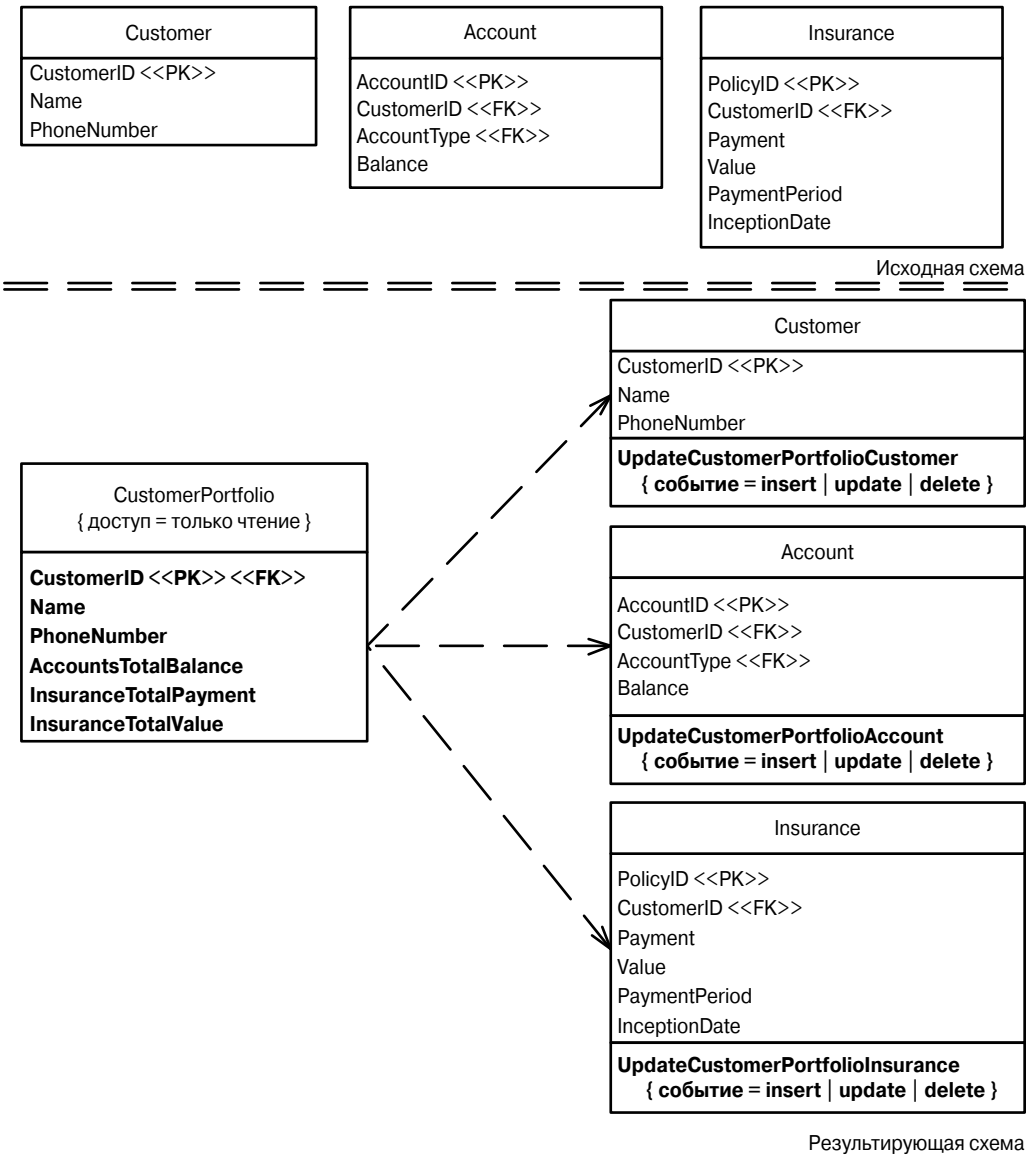


Рис. 9.7. Введение таблицы `CustomerPortfolio`, допускающей только чтение

Обоснование

Необходимость в применении операции “Введение таблицы только для чтения” может быть обусловлена несколькими описанными ниже причинами.

- **Повышение производительности запросов.** Выполнение запроса применительно к какому-то конкретному набору таблиц может оказаться очень медленным в связи с необходимостью в использовании соединений, поэтому применение заранее заполненной таблицы может привести к повышению общей производительности.
- **Накопление итоговых данных для формирования отчетов.** Для многих отчетов требуются итоговые данные, которые можно заранее собрать в таблице, предназначенной только для чтения, а затем многократно использовать.
- **Создание избыточных данных.** Во многих приложениях используются данные, получаемые с помощью запросов из других баз данных в режиме реального времени. А создание в локальной базе данных таблицы, предназначенной только для чтения и содержащей необходимые данные, позволяет снизить зависимость от другой базы (баз) данных, поскольку появляется страховочная копия, которая может использоваться, если доступ к удаленной базе данных прекращается или происходит ее останов для обслуживания.
- **Замена избыточных операций чтения.** Часто наблюдается такая ситуация, что в нескольких внешних программах (или, применительно к рассматриваемой теме, в хранимых процедурах) используются одни и те же запросы выборки данных. Вместо подобных запросов можно перейти к использованию общей таблицы, предназначенной только для чтения, или нового представления; см. описание операции “Введение представления” (с. 325).
- **Защита данных.** При использовании таблицы, допускающей только чтение, конечным пользователям разрешается выполнять запросы к данным, но не обновлять эти данные.
- **Повышение удобства базы данных для чтения.** Если база данных в высшей степени нормализована, то пользователям обычно сложно разобраться в том, какие таблицы из всех возможных должны использоваться для получения требуемой информации. А в результате ввода в действие таблиц, предназначенных только для чтения, в которых собраны широко применяемые, денормализованные структуры данных, схема базы данных становится проще для понимания, поскольку пользователи получают возможность сосредоточиться в своей работе только на денормализованных таблицах.

Потенциальные преимущества и недостатки

Основной проблемой, с которой приходится сталкиваться после ввода в действие таблицы, предназначенной только для чтения, является накопление в ней так называемых “устаревших данных”, т.е. данных, не представляющих текущее состояние исходных данных, на основе которых заполняется рассматриваемая таблица. Например, предположим, что из удаленной системы получены данные и на их основе заполнена локальная таблица, но сразу после этого произошло обновление исходных данных. Из этого следует, что пользователи данных, содержащихся в таблице, предназначенной

только для чтения, должны знать, какова периодичность копирования данных, и учитывать, насколько часто происходят изменения в исходных данных, чтобы определять, приемлемы ли для них данные из этой таблицы, предназначенной только для чтения.

Процедура обновления схемы

Как показано на рис. 9.7, для обновления схемы базы данных при проведении операции "Введение таблицы только для чтения" необходимо вначале создать новую таблицу. Кроме того, нужно принять решение о том, какую структуру будет иметь вводимая в действие таблица, предназначенная только для чтения, затем создать эту таблицу с использованием команды `CREATE TABLE`. После этого необходимо определить стратегию заполнения. Подход, предусматривающий заполнение в реальном времени, должен использоваться, только если конечные пользователи требуют наличия актуальной информации; другими условиями является то, что операция соединения, с помощью которой заполняется таблица, предназначенная только для чтения, является относительно простой, поэтому при проведении операций обновления данных в таблице можно не опасаться сбоев, а также то, что результирующая таблица невелика, поэтому может быть достигнута требуемая производительность. А если вариант с обновлением в реальном времени неприменим, то должен быть принят подход, предусматривающий пакетное обновление.

На рис. 9.7 показан пример, в котором в качестве таблицы, предназначенной только для чтения, используется таблица `CustomerPortfolio`, основанная на таблицах `Customer`, `Account` и `Insurance`, а для ее заполнения служат итоговые данные о деловых операциях, в которых участвуют все клиенты определенной компании. В результате создания таблицы `CustomerPortfolio` для конечных пользователей упрощается задача выполнения произвольных запросов, предназначенных для анализа информации о клиентах. В следующем коде приведены операторы DDL, с помощью которых создается таблица `CustomerPortfolio`:

```
CREATE TABLE CustomerPortfolio (
  CustomerID NUMBER NOT NULL,
  Name VARCHAR(40),
  PhoneNumber VARCHAR2(40),
  AccountsTotalBalance NUMBER,
  InsuranceTotalPayment NUMBER,
  InsuranceTotalValue NUMBER,
  CONSTRAINT PKCustomerPortfolio
    PRIMARY KEY (CustomerID)
);

COMMENT ON CustomerPortfolio 'Read only table for users to query, data
updates will be lost next time this table is refreshed'
```

Для реализации рассматриваемой операции рефакторинга может применяться несколько указанных стратегий. Во-первых, можно оказать доверие разработчикам, рассчитывая на то, что они будут правильно обновлять таблицу, а во всех остальных ситуациях применять ее лишь для выборки данных, для этого обозначить таблицу как предназначенную только для чтения с помощью комментариев. Во-вторых, можно ввести в действие триггеры, которые активизируют сообщение об ошибке при обнаружении попыток вставки, обновления или удаления. В-третьих, можно ограничить

доступ к таблице на основе применяемой стратегии управления защитой доступа (Security Access Control — SAC). В-четвертых, можно использовать операцию “Инкапсуляция таблицы в представление” (с. 266) и обозначить полученное представление как предназначенное только для чтения.

Процедура переноса данных

Для осуществления процедуры переноса данных необходимо скопировать все соответствующие исходные данные в таблицу, предназначенную только для чтения, а затем применить выбранную стратегию заполнения (обновление в реальном времени или периодическое пакетное обновление). Реализация выбранной стратегии заполнения может осуществляться с помощью нескольких стратегий, описанных ниже.

1. **Периодическое обновление.** Применение планируемых заданий, с помощью которых обновляется таблица, предназначенная только для чтения. С помощью такого задания могут обновляться все данные в таблице, предназначенной только для чтения, или обновления могут применяться только для внесения изменений, зафиксированных со времени последнего обновления. Следует отметить, что время, необходимое для обновления данных, должно быть меньше по сравнению с запланированным интервалом времени между обновлениями. Этот метод особенно хорошо подходит для вариантов среды, подобных хранилищам данных, в которых данные за предыдущий день обычно подытоживаются и используются на следующий день. Иными словами, в приложениях допускается использование устаревших данных; кроме того, этот подход открывает возможность применения простого способа синхронизации данных.
2. **Материализованные представления.** В некоторых программных продуктах баз данных обеспечивается возможность использовать представления, которые не возвращают лишь результирующий набор, полученный при выполнении запроса; вместо этого фактически на основе результатов запроса создается таблица. База данных поддерживает такое материализованное представление в актуальном состоянии с учетом опций, выбранных при его создании. Такой метод позволяет использовать встроенные средства базы данных для обновления данных в материализованном представлении, но его основным недостатком может оказаться необходимость применения слишком сложного кода SQL, лежащего в основе представления. Соответствующие программные продукты баз данных действуют таким образом, что при обнаружении чрезмерно сложного кода SQL материализованного представления поддержка автоматизированной синхронизации представления прекращается.
3. **Использование средств синхронизации на основе триггера.** На исходных таблицах создаются триггеры, что позволяет распространять изменения в исходных данных на таблицу, предназначенную только для чтения. Этот метод позволяет применять определяемый пользователем код синхронизации данных, что может потребоваться, если нужно синхронизировать сложные объекты данных, но при этом приходится разрабатывать все необходимые триггеры, что может оказаться трудоемким.

4. **Использование способа обновления в реальном времени с помощью приложения.** Для обновления таблицы, предназначенной только для чтения, может применяться приложение, способное поддерживать данные в актуальном состоянии. Но данный подход осуществим, если известны все приложения, которые записывают данные в исходные таблицы базы данных. Этот метод позволяет использовать для обновления таблицы, предназначенной только для чтения, определенное приложение, благодаря чему данные всегда поддерживаются в актуальном состоянии и можно добиться того, чтобы данные рассматриваемой таблицы не подвергались модификации со стороны других приложений. Недостаток этого подхода состоит в том, что при его использовании запись данных должна осуществляться дважды, вначале в исходную таблицу, а затем в денормализованную, предназначенную только для чтения, что может привести к появлению дублирующихся строк, а связи с этим — программных ошибок.

В следующем коде показано, как можно впервые заполнить данные в таблице `CustomerPortfolio`:

```
INSERT INTO CustomerPortfolio
  SELECT Customer.CustomerID,
    Customer.Name,
    Customer.PhoneNumber,
    SUM(Account.Balance),
    SUM(Insurance.Payment),
    SUM(Insurance.Value)
  FROM
    Customer,Account,Insurance
  WHERE
    Customer.CustomerID=Account.CustomerID
    AND Customer.CustomerID=Insurance.CustomerID
  GROUP BY
    Customer.CustomerID,
    Customer.Name,
    Customer.PhoneNumber;
```

А в приведенном ниже коде показано, как синхронизировать данные, полученные из исходных таблиц `Customer`, `Account` и `Insurance`. Поскольку исходные таблицы являются обновляемыми, а таблица `CustomerPortfolio` — нет, требуется провести только односторонние обновления:

```
CREATE OR REPLACE TRIGGER
UpdateCustomerPortfolioCustomer
  AFTER UPDATE OR INSERT OR DELETE
  ON Customer
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    IF DELETING THEN
      DeleteCustomerPortfolio;
    END IF;
    IF (UPDATING OR INSERTING) THEN
```

```
      UpdateCustomerPortfolio;
    END IF;
  END;
END;
/

CREATE OR REPLACE TRIGGER
UpdateCustomerPortfolioAccount
  AFTER UPDATE OR INSERT OR DELETE
  ON Account
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    UpdateCustomerPortfolioForAccount;
  END;
END;
/

CREATE OR REPLACE TRIGGER
UpdateCustomerPortfolioInsurance
  AFTER UPDATE OR INSERT OR DELETE
  ON Insurance
  REFERENCING OLD AS OLD NEW AS NEW
  FOR EACH ROW
  DECLARE
  BEGIN
    UpdateCustomerPortfolioForInsurance;
  END;
END;
/
```

Процедура обновления программ доступа

Процедура обновления программ доступа осуществляется с учетом причин, по которым такая таблица, предназначенная только для чтения, была введена в действие. Например, как показано на рис. 9.7, если таблица `CustomerPortfolio` должна непосредственно использоваться в приложении, то необходимо обеспечить, чтобы эта таблица действительно рассматривалась в приложении как предназначенная только для чтения. А если `CustomerPortfolio` должна служить в качестве итоговой таблицы для интеграции с некоторыми другими приложениями, то необходимо поддерживать данные в таблице `CustomerPortfolio` в актуальном состоянии. После того как будет принято решение об использовании данных из таблицы `CustomerPortfolio`, необходимо внести изменения во все участки кода, в которых в текущее время осуществляется доступ к исходным таблицам, и предусмотреть использование вместо этого таблицы `CustomerPortfolio`.

Может также потребоваться, чтобы конечным пользователям предоставлялась информация о том, насколько свежими являются данные, путем отображения значений столбца `CustomerPortfolio.LastUpdatedTime`. Кроме того, если для отчетов, формируемых в пакетном режиме, всегда требуются актуальные данные, то пакетное

задание должно быть переработано с тем, чтобы вначале проводилось обновление таблицы CustomerPortfolio, а затем выполнялись отчеты:

```
// Код до рефакторинга
stmt.prepare(
"SELECT Customer.CustomerId, " +
"  Customer.Name, " +
"  Customer.PhoneNumber, " +
"  SUM(Account.Balance) AccountsTotalBalance, " +
"  SUM(Insurance.Payment) InsuranceTotalPayment, " +
"  SUM(Insurance.Value) InsuranceTotalPayment " +
"FROM Customer, Account, Insurance " +
"WHERE " +
"  Customer.CustomerId = Account.CustomerId " +
"  AND Customer.CustomerId = Insurance.CustomerId " +
"  AND Customer.CustomerId = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
"SELECT CustomerId, " +
"  Name, " +
"  PhoneNumber, " +
"  AccountsTotalBalance, " +
"  InsuranceTotalPayment, " +
"  InsuranceTotalPayment " +
"FROM CustomerPortfolio "+
"WHERE CustomerId = ?");
stmt.setLong(1,customer.getCustomerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Операция рефакторинга “Перенос метода из базы данных”

Эта операция позволяет перебазировать существующий метод базы данных (хранимую процедуру, хранимую функцию или триггер) в приложение (приложения), в котором этот метод вызывается в текущий момент (рис. 9.8).

Обоснование

Необходимость в применении операции “Перенос метода из базы данных” может быть обусловлена четырьмя описанными ниже причинами.

- **Проведение доработок, связанных с расширением разнообразия приложений.** Может оказаться так, что при первоначальной реализации метода в базе данных лежащая в его основе бизнес-логика была одинаковой или по крайней мере рассматривалась как одинаковая для всех приложений. Однако со временем требования к приложениям уточнялись, и для отдельных приложений должны применяться различные программные средства доступа к базе данных.

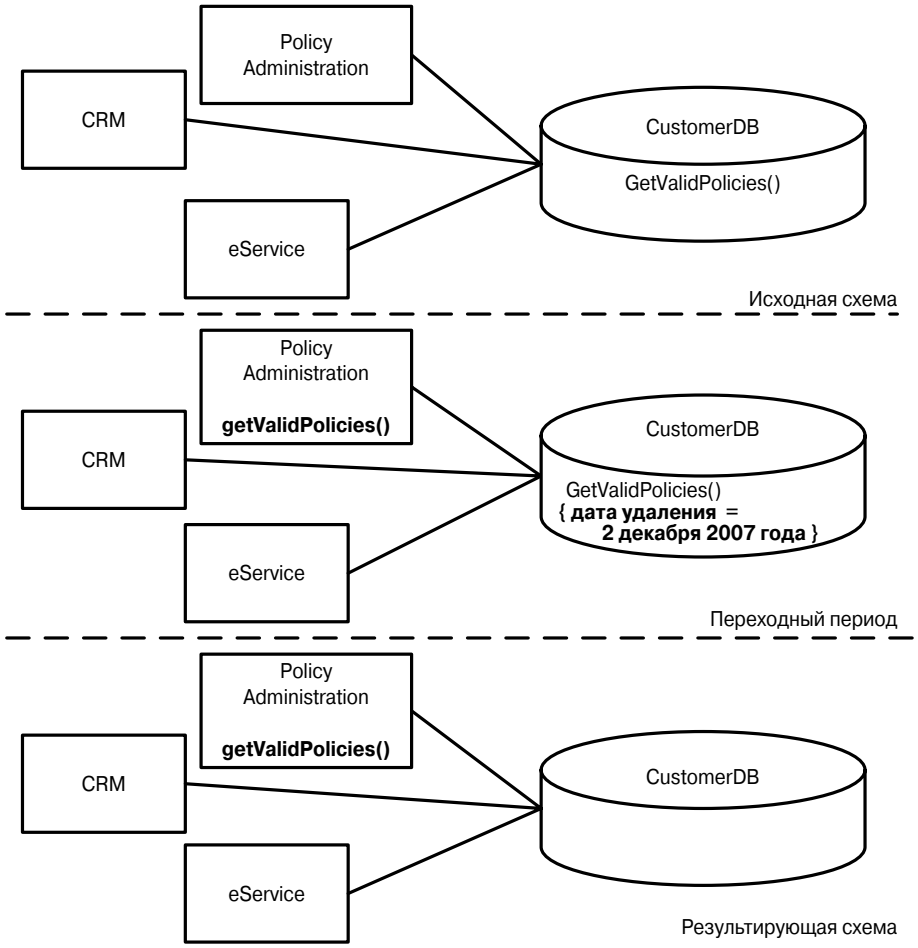


Рис. 9.8. Перемещение кода выборки правил в приложение администрирования правил

- **Обеспечение возможности переноса на другие платформы.** Языки, с помощью которых создаются методы базы данных, имеют отличительные особенности, определяемые отдельными поставщиками баз данных. Поэтому если некоторые программные средства реализуются вне базы данных, возможно, в виде совместно используемого компонента или Web-службы, то появляется возможность упростить задачу переноса этих программных средств в новую базу данных.
- **Увеличение масштабируемости.** Безусловно, в наши дни открывается возможность масштабировать базы данных с помощью Grid-технологии, но никто не может отрицать, что для масштабирования могут также по-прежнему применяться другие средства, такие как ввод в действие дополнительных серверов приложений. Принятая на предприятии стратегия развития архитектуры может предусмат-

ривать масштабирование систем на основе средств, не опирающихся на базы данных, и если в ходе реализации этой стратегии будет обнаружено, что некоторый метод становится узким местом, то может потребоваться вывести его за пределы базы данных.

- **Повышение удобства сопровождения.** Инструментальные средства разработки для таких ведущих языков программирования, как Java, C# и Visual Basic, часто намного сложнее по сравнению с инструментами для работы с языками программирования баз данных. Чем лучше инструментальные средства (а также, разумеется, чем лучше методы разработки), тем проще становится код, а в связи с этим упрощается его сопровождение. Кроме того, если программирование ведется лишь на одном языке, то сокращается потребность в развитии навыков программирования для членов группы проектирования, поскольку легче приобрести, допустим, опыт программирования на языке C#, чем изучить и язык C#, и язык PLSQL, применяемый в СУБД Oracle.

Потенциальные преимущества и недостатки

При проведении рассматриваемой операции рефакторинга необходимо учитывать описанные ниже преимущества и недостатки.

- **Уменьшение возможности повторного использования.** Реляционные базы данных представляют собой технологию, лежащую на самом низком уровне организации работы приложения. Необходимость в использовании баз данных может возникнуть практически в рамках любой прикладной технологии, и поэтому реализация совместно применяемых алгоритмов в реляционной базе данных открывает наибольшие возможности для повторного использования кода.
- **Снижение производительности.** Чем больше данных накапливается в базе данных, тем выше вероятность снижения производительности операций доступа к базе данных, особенно если программный метод организован таким образом, что в нем должен осуществляться доступ к значительным объемам данных, которые должны передаваться в этот метод, и только после этого появляется возможность их обработки.

Процедура обновления схемы

Как показано на рис. 9.8, изменения в схеме базы данных, связанные с осуществлением рассматриваемой операции рефакторинга, являются несложными; достаточно просто обозначить метод как предназначенный для удаления, а затем по истечении переходного периода физически удалить этот метод из базы данных. В рассматриваемом примере хранимая процедура `GetValidPolicies` была перенесена из базы данных `CustomerDB` в приложение администрирования правил. Необходимость в выполнении такой операции обусловлена несколькими причинами. Бизнес-логика, реализованная с помощью процедуры `GetValidPolicies`, требовалась только в одном приложении, сетевой трафик оставался одним и тем же, независимо от того, в каком месте осуществлялась эксплуатация соответствующих функциональных средств, а представители группы проектировщиков решили повысить удобство сопровождения своих

приложений. Любопытно отметить, что немного изменено также имя метода, для того чтобы в нем были отражены соглашения об именовании, принятые в соответствующем языке программирования.

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

В соответствующем приложении (приложениях) должен быть разработан и проверен необходимый метод. В коде программы следует выполнить поиск вызовов существующего метода, а затем заменить этот код вызовами соответствующей функции. В следующем примере кода показано, как должен быть разработан прикладной код для замены функциональных средств, предоставляемых методом:

```
// Код до рефакторинга
stmt.prepareCall("begin ? := getValidPolicies(); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.execute();
ResultSet rs = stmt.getObject(1);
List validPolicies = new ArrayList();
Policy policy = new Policy();
while (rs.next()) {
  policy.setPolicyId(rs.getLong(1));
  policy.setCustomerID(rs.getLong(2));
  policy.setActivationDate(rs.getDate(3));
  policy.setExpirationDate(rs.getDate(4));
  validPolicies.add(policy);
}
return validPolicies;

// Код после рефакторинга
stmt.prepare(
  "SELECT PolicyId, CustomerId, "+
  "ActivationDate, ExpirationDate " +
  "FROM Policy" +
  "WHERE ActivationDate < TRUNC(SYSDATE) AND "+
  "ExpirationDate IS NOT NULL AND "+
  "ExpirationDate > TRUNC(SYSDATE)");
ResultSet rs = stmt.execute();
List validPolicies = new ArrayList();
Policy policy = new Policy();
while (rs.next()) {
  policy.setPolicyId(rs.getLong("PolicyId"));
  policy.setCustomerID(rs.getLong("CustomerId"));
  policy.setActivationDate(rs.getDate("ActivationDate"));
  policy.setExpirationDate(rs.getDate("ExpirationDate"));
  validPolicies.add(policy);
}
return validPolicies;
```

Операция рефакторинга “Перенос метода в базу данных”

Эта операция предназначена для перебазирования существующих прикладных программных средств в базу данных (рис. 9.9).

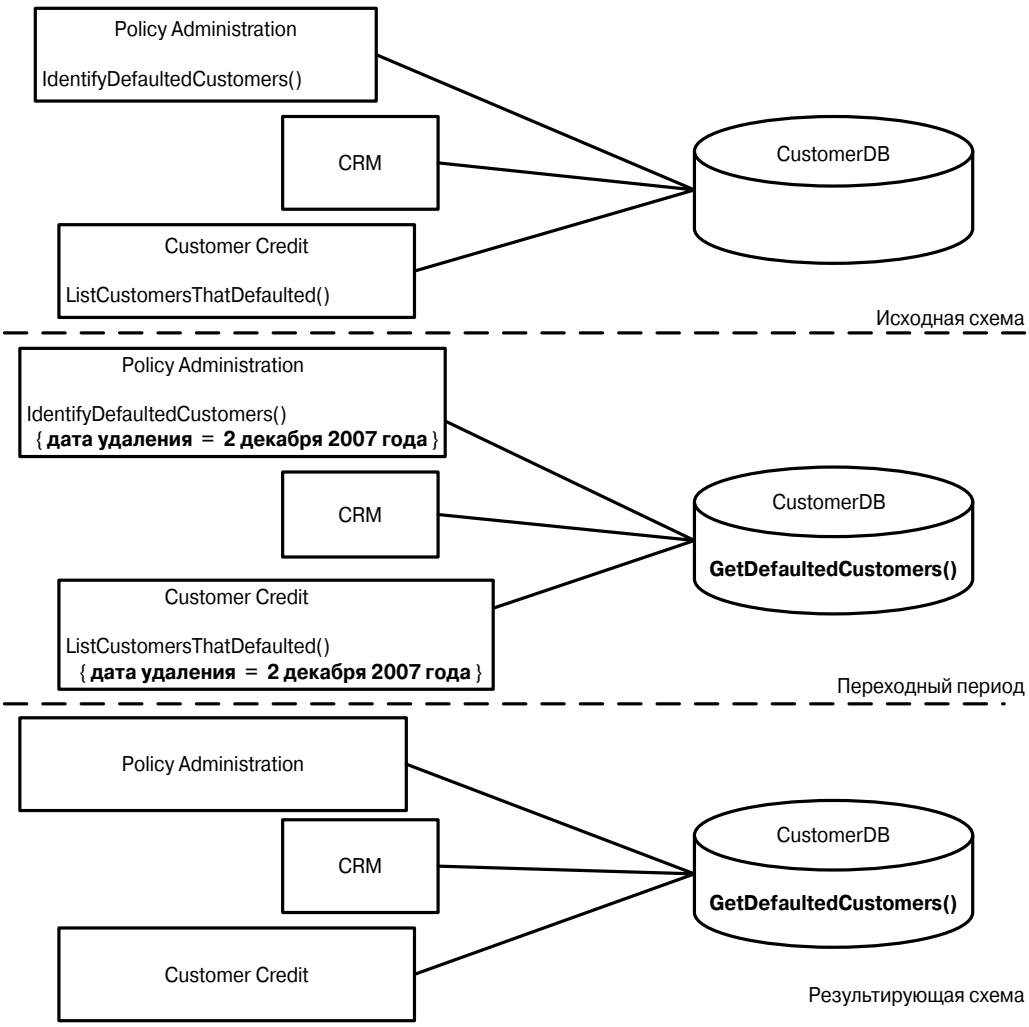


Рис. 9.9. Перенос прикладных программных средств в базу данных

Обоснование

Необходимость в применении операции “Перенос метода в базу данных” может быть обусловлена тремя описанными ниже причинами.

- **Обеспечение повторного использования.** Доступ к реляционным базам данных может быть предусмотрен практически в любой прикладной программной технологии, и поэтому реализация в реляционной базе данных таких совместно используемых программных средств, как методы базы данных (хранимые процедуры, функции или триггеры), обеспечивает наибольшие возможности для их многократного применения.
- **Увеличение масштабируемости.** Безусловно, в наши дни открывается возможность масштабировать базы данных с помощью Grid-технологии, поэтому на основе принципа базирования программных средств обработки данных непосредственно в базе данных может быть создана вся архитектура программного обеспечения предприятия.
- **Повышение производительности.** Применение хранимых процедур открывает возможность повышения производительности, особенно если с ее помощью выполняется обработка значительных объемов данных, поскольку обработка осуществляется на том же компьютере, где эксплуатируется база данных, что способствует уменьшению объема результирующих наборов, передаваемых по сети.

Потенциальные преимущества и недостатки

Основным недостатком рассматриваемой операции рефакторинга является уменьшение переносимости методов базы данных, поскольку обычно в коде этих методов применяются программные конструкции, характерные для конкретных баз данных, разрабатываемых отдельными поставщиками баз данных. Но эту проблему во многом преувеличивают специалисты, далекие от практики, поскольку организации крайне редко переходят от продуктов одного поставщика баз данных к продуктам другого, возможно, потому, что поставщики находят способы привязать к себе существующую базу клиентов, или потому, что необходимые для этого расходы часто бывают неоправданными.

Процедура обновления схемы

Как показано на рис. 9.9, соответствующие изменения в схеме базы данных являются несложными; достаточно лишь разработать и проверить хранимую процедуру. В этом примере программные средства применяются для получения списка недобросовестных клиентов, которые не вносят вовремя свои платежи. Эти программные средства реализованы в приложениях Policy Administration и Customer Credit. В каждом приложении рассматриваемая операция названа по-разному, соответственно `IdentifyDefaultedCustomers()` и `ListCustomersThatDefaulted()`, хотя обе эти функции могут быть заменены одной — `CustomerDB.GetDefaultedCustomers()`.

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

Программное обеспечение метода должно быть удалено из всех реализующих его приложений, а вместо этого необходимо предусмотреть вызов хранимой процедуры. Наиболее простой способ достижения этой цели состоит в том, чтобы вызывать хранимую процедуру из существующего прикладного метода (методов). Может быть также обнаружено, что в разных приложениях такой метод назван по-разному или даже, возможно, что соответствующие методы отличаются друг от друга по своему проекту. Например, предположим, что в одном приложении вся бизнес-логика реализована в виде одной большой функции, а в другом представлена как несколько меньших функций.

Аналогичным образом, может быть обнаружено, что в различных приложениях рассматриваемый метод реализован по-разному, либо потому, что код больше не отражает фактические требования (если только когда-либо их отражал), либо потому, что имеются весомые основания для того, чтобы соответствующие программные средства были действительно реализованы в отдельных приложениях по-разному. Например, допустим, что операция `DetermineVacationDays(Year)` позволяет получить список рабочих дней, за которые сотрудникам начисляется оплаченный отпуск. Эта операция реализована в нескольких приложениях, но после изучения кода этих приложений обнаруживается, что она реализована в них по-разному. Различные версии были написаны в разное время, а со времени написания более старых версий приложений правила изменились так, что в них теперь отражены разные требования, выдвинутые государством, штатом, а иногда даже городом. После обнаружения подобной ситуации может быть принято решение оставить различные версии нетронутыми (например, смириться с различиями), исправить существующие прикладные методы или написать и ввести в базу данных хранимую процедуру (процедуры), в которой реализована правильная версия программных средств.

В следующем примере кода показано, как можно переместить прикладные программные средства в базу данных и использовать после этого хранимую процедуру в приложении для получения списка задолжавших клиентов. В этом примере не показан код создания хранимой процедуры:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT CustomerId, "+
  "PaymentAmount" +
  "FROM Transactions" +
  "WHERE LastPaymentdate < TRUNC(SYSDATE-90) AND "+
  "PaymentAmount > 30 ");
ResultSet rs = stmt.execute();
List defaulters = new ArrayList();
DefaultedCustomer defaulted  = new DefaultedCustomer();
while (rs.next()) {
  defaulted.setCustomerID(rs.getLong("CustomerId"));
  defaulted.setAmount(rs.getBigDecimal("PaymentAmount"));
```

```
  defaulters.add(defaulted);
}
return defaulters;

// Код после рефакторинга
stmt.prepareCall("begin ? := getDefaultedCustomers(); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.execute();
ResultSet rs = stmt.getObject(1);
List defaulters = new ArrayList();
DefaultedCustomer defaulted  = new DefaultedCustomer();
while (rs.next()) {
  defaulted.setCustomerID(rs.getLong(1));
  defaulted.setAmount(rs.getBigDecimal(2));
  defaulters.add(defaulted);
}
return defaulters;
```

Операция рефакторинга “Замена метода (методов) представлением”

Эта операция позволяет создать представление, основанное на одном или нескольких существующих методах базы данных (хранимых процедурах, хранимых функциях или триггерах), которые до сих пор вызывались из базы данных (рис. 9.10).

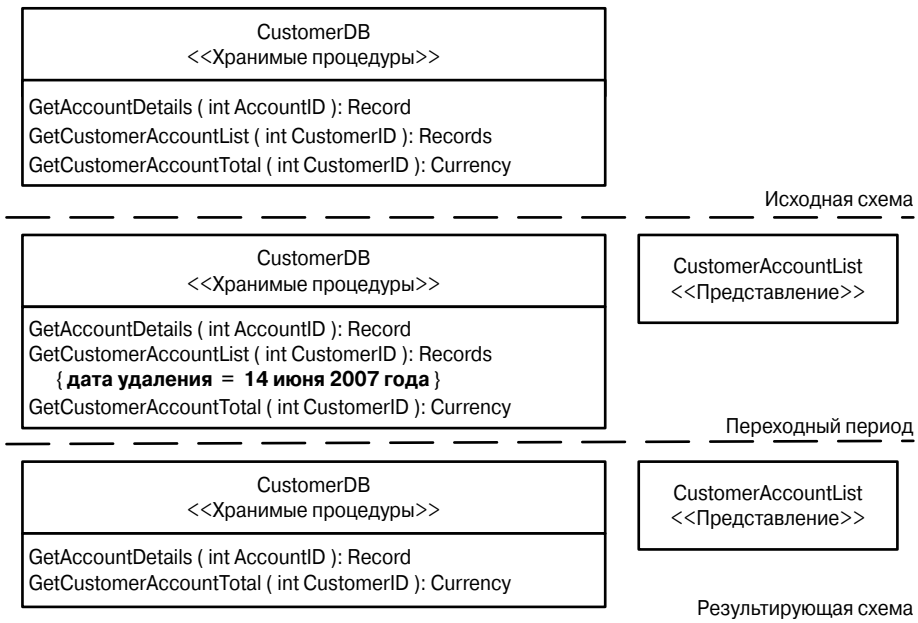


Рис. 9.10. Ввод в действие представления `CustomerAccountList`

Обоснование

Необходимость в применении операции "Замена метода (методов) представлением" может быть обусловлена тремя описанными ниже основными причинами.

- **Повышение удобства в эксплуатации.** Возможно, что внедрены новые инструментальные средства, для работы с которыми гораздо удобнее использовать представления, а не методы; такой особенностью обладают, в частности, инструментальные средства формирования отчетов.
- **Уменьшение объема сопровождения.** Практика показывает, что можно проще обеспечить сопровождение представлений по сравнению с методами.
- **Обеспечение переносимости.** В результате применения представлений переносимость схемы базы данных увеличивается. Для создания методов базы данных используются языки, которые имеют определенные отличия, зависящие от версий программных продуктов различных поставщиков баз данных, тогда как определения представлений могут быть написаны так, чтобы они были совместимыми со стандартами SQL.

Потенциальные преимущества и недостатки

Эта операция рефакторинга, как правило, может применяться только к относительно простым методам, реализующим программные средства, которые могут быть также реализованы с помощью определения представления. Из этого следует, что рассматриваемая операция рефакторинга ограничивает возможности выбора архитектуры программного обеспечения. Кроме того, если используемая база данных не поддерживает обновляемые представления, то приходится ограничиваться заменой только методов, предназначенных для выборки данных. Выполнение этой операции рефакторинга практически не отражается на производительности и масштабируемости, поскольку все действия по-прежнему осуществляются непосредственно в самой базе данных.

Процедура обновления схемы

Прежде чем приступать к обновлению схемы базы данных, следует ввести в действие необходимое представление с помощью операции рефакторинга "Введение представления" (с. 325). После этого нужно отметить соответствующий метод как предназначенный для удаления и в конечном итоге удалить его с помощью команды `DROP PROCEDURE` по истечении переходного периода. На рис. 9.10 показан пример, в котором хранимая процедура `GetCustomerAccountList` заменяется представлением `CustomerAccountList`. А в следующем коде приведены операторы DDL, предназначенные для выполнения этого действия:

```
CREATE VIEW CustomerAccountList (
  CustomerID     NUMBER NOT NULL,
  CustomerName   VARCHAR(40),
  CustomerPhone  VARCHAR2(40),
  AccountNumber  VARCHAR(14),
  AccountBalance NUMBER,
) AS
```

```
SELECT
  Customer.CustomerID,
  Customer.Name,
  Customer.PhoneNumber,
  Account.AccountNumber,
  Account.Balance
FROM Customer,Account
WHERE Customer.CustomerID=Account.CustomerID
;

-- Выполнить этот код после 14 июня 2007 года
DROP PROCEDURE GetCustomerAccountList
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

Должны быть подвергнуты рефакторингу программы доступа, для того чтобы они работали с представлением, а не вызывали хранимую процедуру. После перехода к использованию представления коды ошибок, активизируемых базой данных, изменяются, поэтому может потребоваться переопределить код обработки ошибок. В следующем коде показано, как внести изменения в приложение для вызова представления, а не хранимой процедуры:

```
// Код до рефакторинга
stmt.prepareCall("begin ? := getCustomerAccountList(?); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.setInt(1,customerId);
stmt.execute();
ResultSet rs = stmt.getObject(1);
List customerAccounts = new ArrayList();
while (rs.next()) {
  customerAccounts.add(populateAccount(rs));
}
return customerAccounts;

// Код после рефакторинга
stmt.prepare(
  "SELECT CustomerID, CustomerName, "+
  "CustomerPhone, AccountNumber, AccountBalance " +
  "FROM CustomerAccountList " +
  "WHERE CustomerId = ? ");
stmt.setLong(1,customerId);
ResultSet rs = stmt.executeQuery();
List customerAccounts = new ArrayList();
while (rs.next()) {
  customerAccounts.add(populateAccount(rs));
}
return customerAccounts;
```

Операция рефакторинга “Замена представления методом (методами)”

Эта операция позволяет заменить существующее представление одним или несколькими существующими методами (хранимыми процедурами, хранимыми функциями или триггерами), находящимися в базе данных (рис. 9.11).

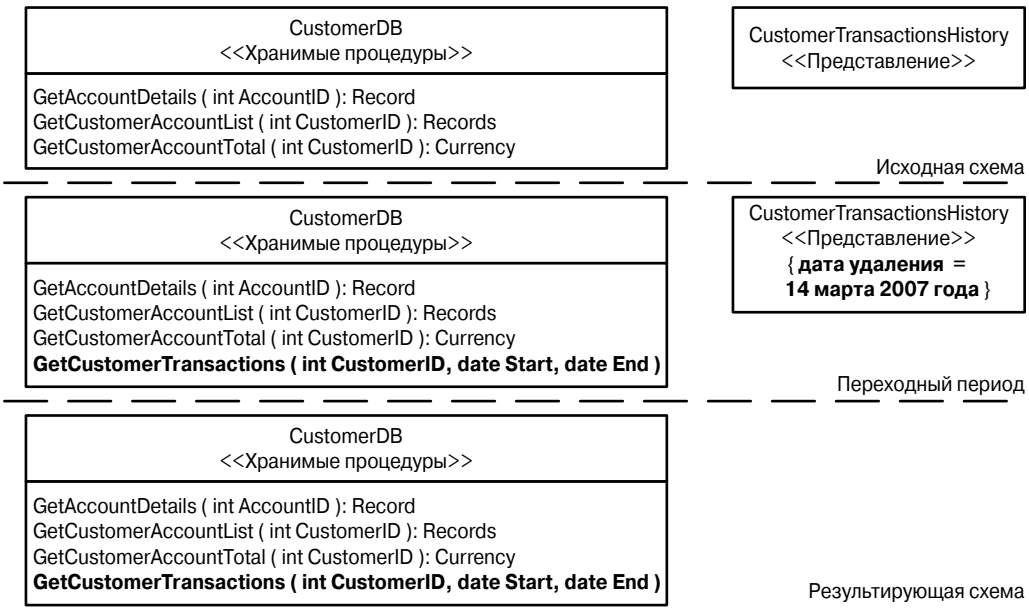


Рис. 9.11. Ввод в действие хранимой процедуры `GetCustomerTransactions`

Обоснование

Необходимость в применении операции “Замена представления методом (методами)” может быть обусловлена двумя причинами. Во-первых, методы позволяют реализовать более сложные программные средства по сравнению с представлениями, поэтому рассматриваемая операция рефакторинга может стать первым шагом в этом направлении. Во-вторых, методы позволяют обновлять таблицы с данными. В некоторых базах данных обновляемые представления не поддерживаются или чаще всего ограничиваются возможностями обновления единственной таблицы. Поэтому перемещение средств обновления данных из представлений в методы открывает возможность создания более сложной архитектуры базы данных.

Потенциальные преимущества и недостатки

При использовании рассматриваемой операции рефакторинга могут возникать некоторые проблемы, описанные ниже. Во-первых, инструментальные средства формирования отчетов обычно более успешно работают с представлениями, а не с методами.

Во-вторых, переход к использованию методов может привести к снижению удобства переносимости программного обеспечения, поскольку особенности языков, применяемых для разработки методов базы данных, изменяются в зависимости от версий, разрабатываемых отдельными поставщиками баз данных. В-третьих, может снизиться удобство сопровождения, поскольку некоторые специалисты предпочитают работать с представлениями, а не с методами (и наоборот). К счастью, производительность и масштабируемость редко испытывают отрицательное влияние, поскольку вся работа по-прежнему происходит в базе данных.

Процедура обновления схемы

Прежде чем приступать к обновлению схемы базы данных, необходимо ввести соответствующий метод (методы), требуемый для замены представления. Затем необходимо отметить представление как предназначенное для удаления и в конечном итоге его удалить по истечении переходного периода с помощью операции рефакторинга “Удаление представления” (с. 118).

На рис. 9.11 показан пример, в котором представление `CustomerTransactionsHistory` заменяется хранимой процедурой `GetCustomerTransactions`. В следующем коде приведены операторы, позволяющие ввести в действие метод и удалить представление:

```
CREATE OR REPLACE PROCEDURE GetCustomerTransactions
{
  P_CustomerID IN NUMBER
  P_Start IN DATE
  P_End IN DATE
}
IS
BEGIN
  SELECT *
  FROM Transaction
  WHERE Transaction.CustomerID = P_CustomerID
  AND Transaction.PostingDate BETWEEN P_Start AND P_End;
END;

-- 14 марта 2007 года
DROP VIEW CustomerTransactionsHistory;
```

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

Должны быть подвергнуты рефакторингу программы доступа, чтобы они могли выполнять свою работу с помощью метода (методов), а не представления. После перехода к использованию методов коды ошибок, активизируемых базой данных, изменяются, поэтому может потребоваться переопределить код обработки ошибок.

После замены представления CustomerTransactionsHistory хранимой процедурой GetCustomerTransactions в код необходимо внести изменения, как показано ниже.

```
// Код до рефакторинга
stmt.prepare(
  "SELECT * " +
  "FROM CustomerTransactionsHistory " +
  "WHERE CustomerId = ? "+
  "AND TransactionDate BETWEEN ? AND ? ");
stmt.setLong(1,customerId);
stmt.setDate(2,startDate);
stmt.setDate(3,endDate);
ResultSet rs = stmt.executeQuery();
List customerTransactions = new ArrayList();
while (rs.next()) {
  customerTransactions.add(populateTransactions(rs));
}
return customerTransactions;

// Код после рефакторинга
stmt.prepareCall("begin ? := getCustomerTransactions(?,?,?); end;");
stmt.registerOutParameter(1, OracleTypes.CURSOR);
stmt.setInt(1,customerId);
stmt.setDate(2,startDate);
stmt.setDate(3,endDate);
stmt.execute();
ResultSet rs = stmt.getObject(1);
List customerTransactions = new ArrayList();
while (rs.next()) {
  customerTransactions.add(populateAccount(rs));
}
return customerTransactions;
```

Операция рефакторинга “Использование официально заданного источника данных”

Эта операция позволяет перейти к использованию официально утвержденного источника данных для данной конкретной сущности вместо того источника данных, который используется в настоящее время (рис. 9.12 и 9.13).

Обоснование

Необходимость в применении операции “Использование официально заданного источника данных” может быть в основном обусловлена требованием использовать правильную версию данных для рассматриваемой таблицы (или таблиц). Если одни и те же данные хранятся в нескольких местах, то возникает риск появления несовместимости и нарушения доступа. Например, предположим, что на предприятии имеется несколько баз данных, одна из которых, база данных CRM, является официальным источником информации о клиентах. Если же в приложении используется его собственная таблица Customer, то может оказаться так, что в этом приложении для работы не привлекаются

все данные о клиентах, имеющиеся в конкретной организации. Еще худшая ситуация может возникнуть, если в приложении регистрируются данные о новом клиенте, но эта информация не становится доступной тем, кто пользуется базой данных CRM, и поэтому не предоставляется для других приложений, эксплуатируемых в организации.

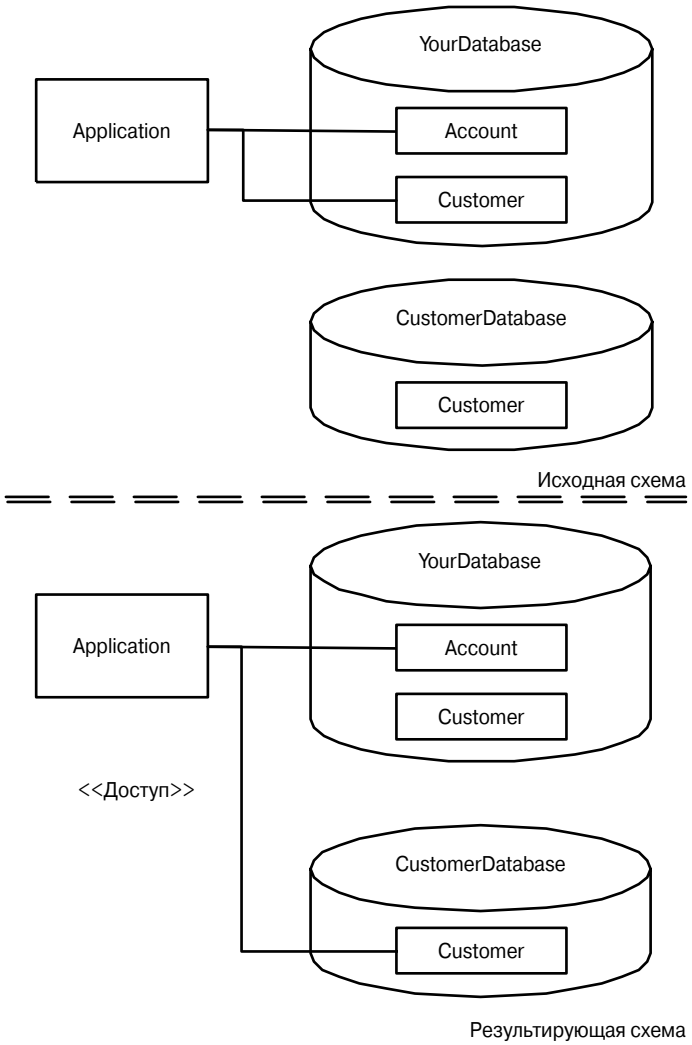


Рис. 9.12. Обеспечение непосредственного доступа к официально утвержденным данным о клиентах из базы данных `Customer`

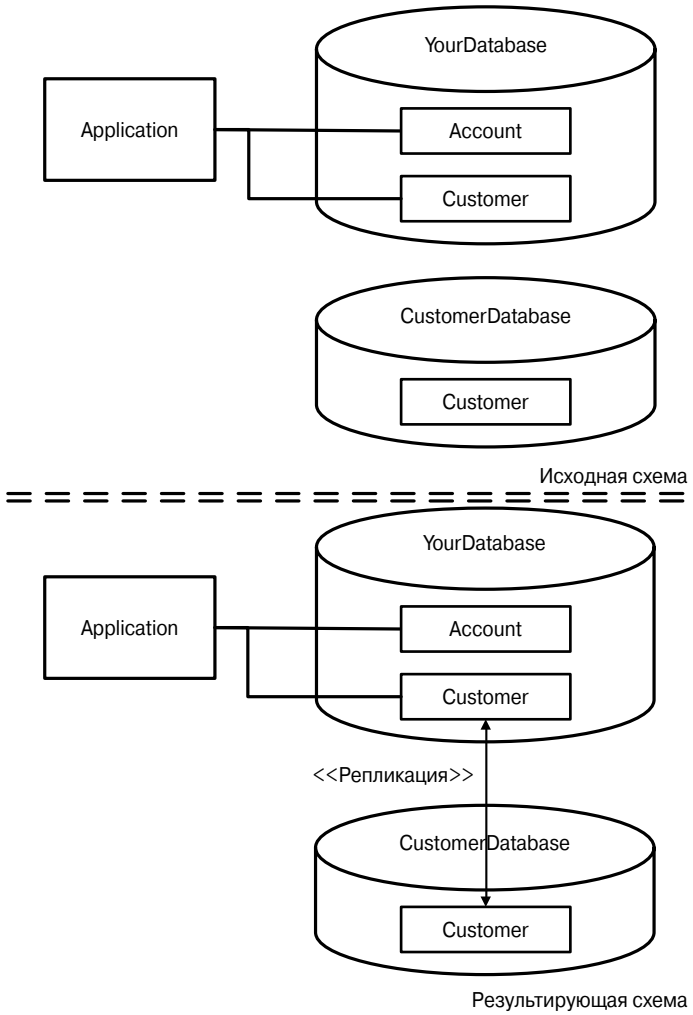


Рис. 9.13. Переход к использованию официально утвержденных данных о клиенте с помощью стратегии репликации

Потенциальные преимущества и недостатки

При оценке основных потенциальных преимуществ и недостатков необходимо учитывать, оправдан ли переход к использованию официально утвержденных таблиц базы данных, если учесть все затраты, связанные с рефакторингом всех ссылок на таблицы в локальной базе данных. А если стратегия поддержки ключей в официальном источнике данных отличается от той, которая была принята при создании конкретной таблицы, то может потребоваться вначале применить операцию "Осуществление стратегии консолидированных ключей" (с. 197), чтобы идентификаторы, используемые в базе данных, соответствовали идентификаторам, принятым в официально утвержденном

источнике данных. Кроме того, данные в официальном источнике данных могут иметь другой смысл и характеризоваться другой периодичностью обновления по сравнению с теми данными, которые используются в настоящее время. В связи с этим для преобразования существующих данных в данные, которые соответствуют официально утвержденному источнику данных, может потребоваться применить такие операции рефакторинга, как “Применение стандартных кодовых обозначений” (с. 188), “Применение стандартного типа” (с. 192) и “Введение общего формата” (с. 210).

Процедура обновления схемы

Для обновления схемы необходимо выполнить описанные ниже действия.

1. **Определить официально утвержденный источник данных.** Прежде всего необходимо определить официально утвержденный источник данных, которым часто становится внешняя база данных, выходящая за пределы вашего контроля. Еще худшая ситуация возникает, если имеется несколько “официально утвержденных источников данных”, которые приходится либо использовать выборочно, либо консолидировать предоставляемые ими данные с учетом конкретной ситуации. В связи с этим необходимо заключить соглашение с лицом (лицами), заинтересованным в разработке проекта, и с владельцами источника данных, к которому приложение должно получить доступ.
2. **Выбрать стратегию реализации.** Для решения этой задачи можно воспользоваться одним из двух вариантов: переработать приложение (приложения) в целях непосредственного получения доступа к официально утвержденному источнику данных (пример осуществления такой стратегии показан на рис. 9.10) или организовать репликацию исходных данных в одной из существующих таблиц (пример реализации такой стратегии приведен на рис. 9.11). Если официально утвержденный источник данных находится в другой базе данных, то стратегия репликации обеспечивает лучшее масштабирование по сравнению со стратегией непосредственного доступа, поскольку открывает возможность ограничиться привязкой приложения только к одной базе данных.
3. **Реализовать стратегию непосредственного доступа.** В приложение должны быть внесены такие изменения, чтобы оно получало непосредственный доступ к базе данных, в рассматриваемом случае к базе данных `CustomerDatabase`. Следует отметить, что для этого может потребоваться другое соединение с базой данных, а не то, которое используется в текущее время. Но осуществление этой стратегии дает возможность выполнять транзакции, охватывающие несколько разных баз данных.
4. **Реализовать стратегию репликации.** Можно организовать применение средств репликации, которые охватывают две базы данных, допустим, `YourDatabase` и `CustomerDatabase`, чтобы с их помощью осуществлялась репликация всех необходимых таблиц. Если требуется обеспечить наличие обновляемых таблиц в своей базе данных, `YourDatabase`, необходимо использовать репликацию с несколькими ведущими. В результате осуществления этой стратегии не потребуется вносить изменения в прикладной код, при условии, что схема и семантика исходных данных остаются одинаковыми. Если требуется внесение изменений в

схему, таких как переименовывание таблиц, или, что еще хуже, если семантика официально утвержденных данных отличается от семантики текущих данных, то, скорее всего, потребуется внести изменения во внешние программы.

5. **Удалить таблицы, которые больше не используются.** Если будет решено обеспечить непосредственный доступ к официально утвержденным исходным данным, то соответствующие таблицы в базе данных `YourDatabase` больше не потребуются. Для удаления этих таблиц необходимо воспользоваться операцией рефакторинга "Удаление таблицы" (с. 117).

Процедура переноса данных

При использовании этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют, если семантика данных официально утвержденной базы данных `CustomerDatabase` и локальной базы данных `YourDatabase` является одинаковой. Если же семантика данных в этих двух базах данных различается, то необходимо либо предусмотреть возможность отказа от этого варианта рефакторинга и перейти к использованию стратегии репликации, в которой осуществляется прямое и обратное преобразование различающихся значений, либо осуществить рефакторинг приложения (приложений), для того, чтобы в нем учитывалась семантика данных официально утвержденного источника данных. Если между данными нет семантического подобия или не удалось найти удобный способ прямого и обратного преобразования данных, то, скорее всего, стратегия репликации также окажется неприменимой.

Процедура обновления программ доступа

Способ внесения изменений во внешние программы изменяется в зависимости от выбранной стратегии реализации. Если используется стратегия непосредственного доступа, то в программах необходимо предусмотреть подключение к официально утвержденному источнику данных, в рассматриваемом случае к базе данных `CustomerDatabase`, и обеспечить работу с этими данными. Если же применяется стратегия репликации, то необходимо создать сценарии репликации, которые связывают соответствующие таблицы баз данных `YourDatabase` и `CustomerDatabase`. А если таблица (таблицы) в базе данных `YourDatabase` совпадает по своей структуре с таблицей (таблицами) базы данных `CustomerDatabase`, то необходимость внесения изменений в программы отсутствует. В противном случае необходимо либо принять за основу схему и семантику официально утвержденного источника данных, а затем подвергнуть рефакторингу приложение (приложения) должным образом, либо найти способ осуществления репликации таким образом, чтобы схема базы данных `YourDatabase` не изменялась. Один из вариантов решения такой задачи состоит в подготовке сценариев, выполняющих прямое и обратное преобразование, что чаще всего становится затруднительным для осуществления на практике, поскольку отображения между данными почти никогда не бывают взаимно однозначными. Еще один вариант, столь же сложный, предусматривает замену исходной таблицы (таблиц) в базе данных `YourDatabase` соответствующими таблицами из базы данных `CustomerDatabase`, с последующим применением операции рефакторинга "Инкапсуляция таблицы в представление" (с. 266), в результате чего существующие приложения получат возможность по-прежнему работать со "старой

схемой”. В этом новом представлении (представлениях) необходимо будет реализовать такие же программные средства преобразования, как и в упомянутых ранее сценариях.

В следующем коде показано, как внести изменения в приложение для обеспечения работы с официально утвержденным источником данных и перехода к использованию соединения `crmDB` вместо прежнего соединения:

```
// Код до рефакторинга
stmt = DB.prepare("select CustomerID,Name,PhoneNumber "+
  "FROM Customer WHERE CustomerID = ?");
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();

// Код после рефакторинга
stmt = crmDB.prepare("select CustomerID,Name,PhoneNumber "+
  "FROM Customer WHERE CustomerID = ?");
stmt.setLong(1, customerID);
stmt.execute();
ResultSet rs = stmt.executeQuery();
```

Глава 10

Операции рефакторинга методов

В настоящей главе приведены общие сведения об операциях рефакторинга кода, которые авторы нашли применимыми к хранимым процедурам, хранимым функциям и триггерам. Для упрощения эти три типа функциональных средств упоминаются просто как методы, в соответствии с терминологией, которую Мартин Фаулер использовал в своей книге *Refactoring* [**17**]. Как указывает само это название, операция рефакторинга метода представляет собой изменение в хранимой процедуре, способствующее повышению ее качества. Авторы поставили перед собой цель представить краткий обзор этих операций рефакторинга. А более подробное описание таких операций можно найти в книге *Refactoring* — в оригинальной работе на эту тему.

Авторы проводят различие между двумя категориями операций рефакторинга методов, поэтому ниже отдельно рассматриваются операции, предусматривающие внесение изменений в интерфейс, предоставляемый базой данных для внешних программ, и операции, которые изменяют внутреннюю организацию метода. Например, применение операции рефакторинга "Добавление параметра" (с. 300) приводит к изменению интерфейса метода, тогда как применение операции "Консолидация условного выражения" (с. 305) не влечет за собой такие последствия. (Выполнение последней операции приводит к замене сложного оператора сравнения вызовом нового метода, который реализует только это сравнение.) Мы проводим различие между этими двумя категориями, поскольку после проведения операций рефакторинга, которые изменяют интерфейс, возникает необходимость подвергать также рефакторингу внешние программы, а в случае применения операций, модифицирующих только внутреннюю организацию метода, такая необходимость отсутствует.

10.1. Операции рефакторинга, которые приводят к изменению интерфейса

В каждой из этих операций рефакторинга предусмотрен переходный период, который предоставляет группе разработчиков достаточно времени для обновления внешних программ с вызовами соответствующих методов. Часть такой операции рефакторинга состоит в переработке исходной версии метода (методов) для обеспечения вызова новой версии (версий) должным образом, а затем, после завершения переходного периода, вызывается соответствующим образом старая версия.

Операция рефакторинга “Добавление параметра”

Как показывает само название рассматриваемой операции рефакторинга, в ней предусматривается добавление нового параметра к существующему методу. На рис. 10.1 приведен пример, в котором к методу `ReadCustomer` добавлен третий параметр, `MiddleName`. Наиболее безопасным способом добавления параметра является его присоединение к концу списка параметров; благодаря этому при проведении рефакторинга вызовов метода не возникает риск того, что параметры будут переупорядочены не в соответствии с их расположением в прототипе метода. Если же в дальнейшем будет решено изменить порядок расположения параметров в прототипе метода, то можно применить операцию рефакторинга “Переупорядочение параметров” (с. 302).

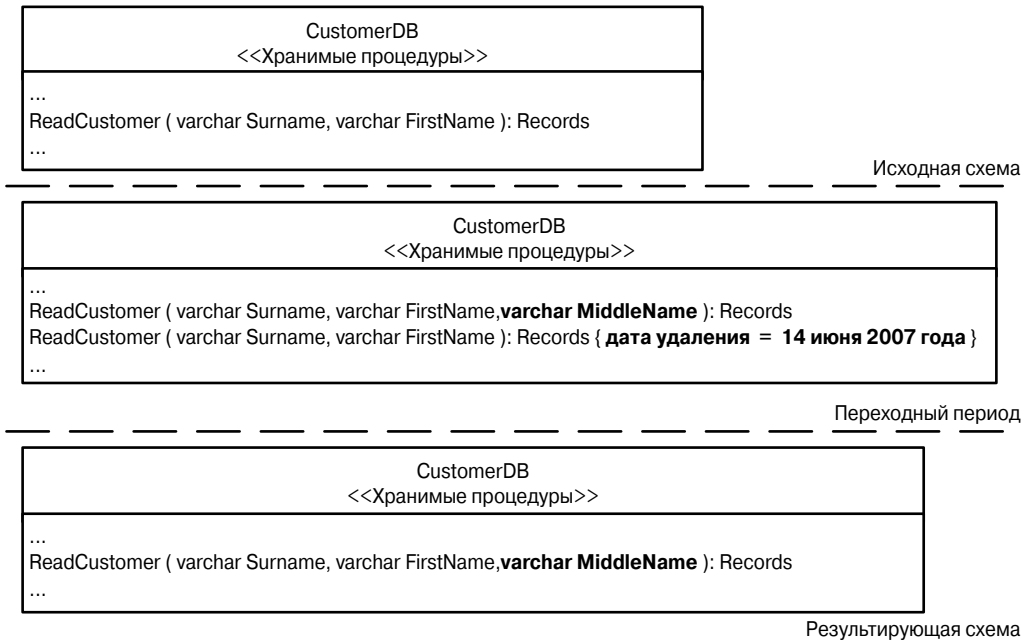


Рис. 10.1. Добавление параметра

Операция рефакторинга “Параметризация метода”

Иногда обнаруживается, что в двух или нескольких методах выполняются, по существу, одни и те же действия. Например, как показано на рис. 10.2, хранимые процедуры `GetAmericanCustomers`, `GetCanadianCustomers` и `GetBrazilianCustomers` формируют списки клиентов из Соединенных Штатов, Канады и Бразилии. Безусловно, применение этих отдельных хранимых процедур связано с определенным преимуществом, которое заключается в том, что они специализированы, поэтому обеспечивают выполнение правильных запросов к базе данных, но задача их сопровождения сложнее по сравнению с сопровождением единственной хранимой процедуры. В рассматриваемом примере эти три хранимые процедуры заменяются хранимой процедурой `GetCustomerByCountry`, которая

принимает в качестве параметра идентификатор страны. В результате объем кода, требующего сопровождения, уменьшается, кроме того, становится проще обеспечивать включение в приложения информации о новых государствах, поскольку для этого достаточно ввести новые коды государств.

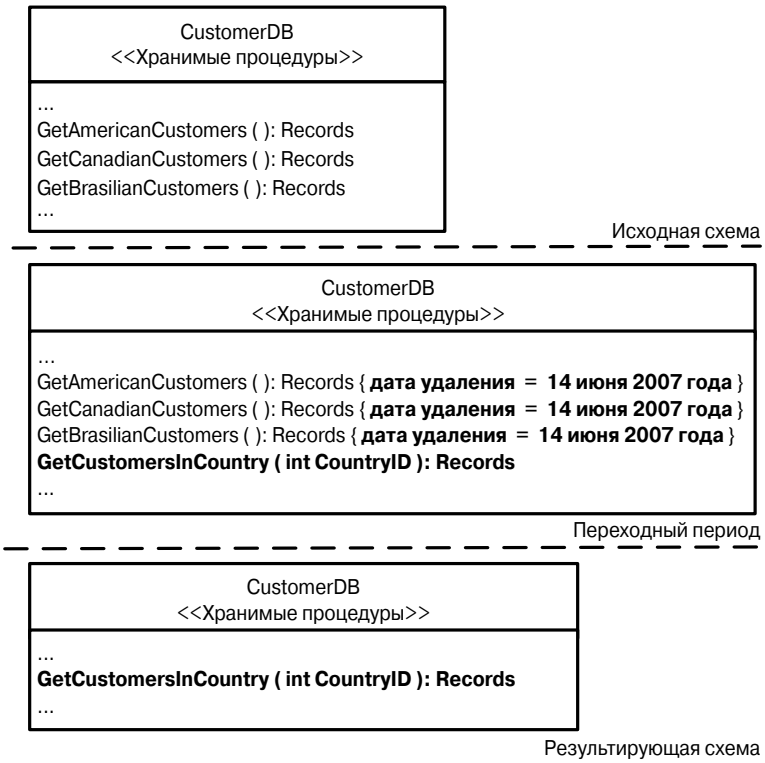


Рис. 10.2. Параметризация хранимой процедуры

Операция рефакторинга "Удаление параметра"

Иногда возникает такая ситуация, что метод включает лишний параметр, который был предусмотрен первоначально, возможно, в расчете на то, что он потребуется в будущем, или, возможно, стал лишним потому, что изменились деловые требования и необходимость в нем исчезла, или же в связи с тем, что появилась возможность получить значение этого параметра другим способом, таким как чтение этого значения из поисковой таблицы. Как показано на рис. 10.3, в вызове метода `GetAccountList` предусмотрен ненужный параметр `AsOfDate`, который должен быть удален, для того чтобы вызов хранимой процедуры соответствовал фактическому назначению этой процедуры.

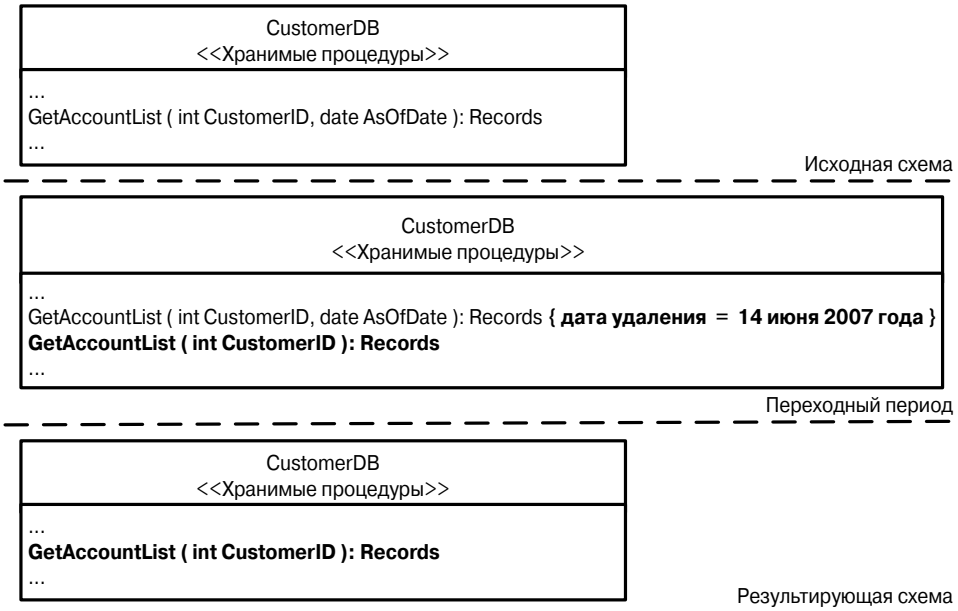


Рис. 10.3. Удаление параметра

Операция рефакторинга "Переименование метода"

Иногда обнаруживается, что метод имеет неудачное имя или это имя не соответствует корпоративным соглашениям об именовании. Например, на рис. 10.4 показано, как переименовать хранимую процедуру `GetAccountList`, присвоив ей имя `GetAccountsForCustomer`, чтобы привести имя процедуры в соответствие с принятыми стандартными соглашениями об именовании.

Операция рефакторинга "Переупорядочение параметров"

В процессе эксплуатации часто обнаруживается, что существующее упорядочение параметров метода не имеет смысла, возможно потому, что с помощью операции "Добавление параметра" (с. 300) были добавлены новые параметры в неправильном порядке или просто по той причине, что упорядочение параметров перестало соответствовать текущим деловым потребностям. Так или иначе, предусмотрена возможность упростить понимание назначения хранимой процедуры путем упорядочения ее параметров должным образом (рис. 10.5).

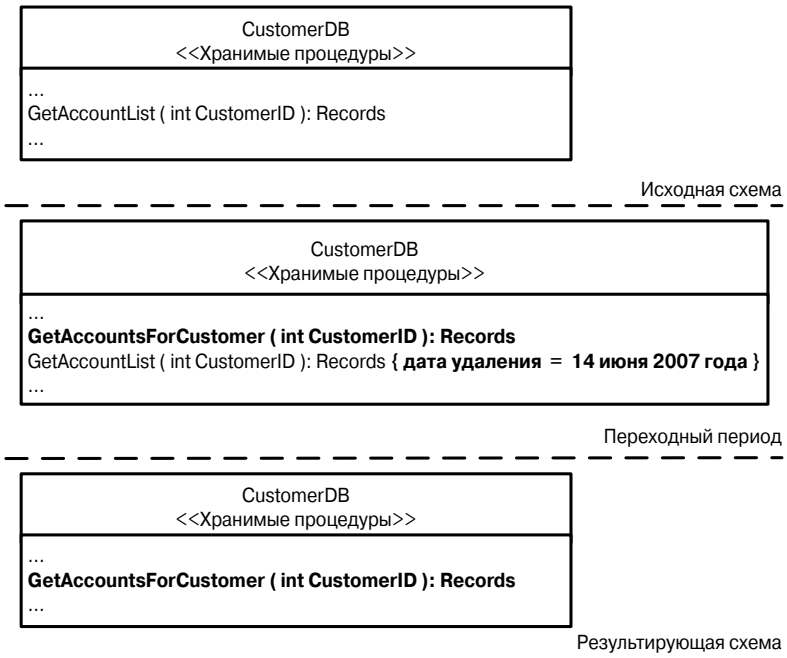


Рис. 10.4. Переименование хранимой процедуры

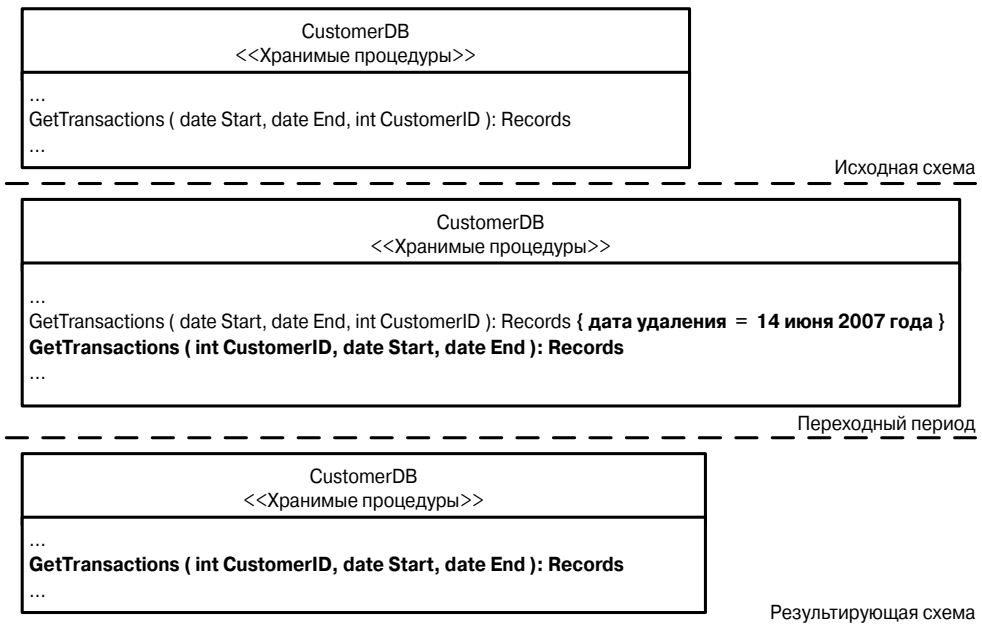


Рис. 10.5. Переупорядочение параметров метода

Как показано на рис. 10.6, параметры метода `GetCustomers` были переупорядочены так, чтобы они отражали соглашения об упорядочении параметров, принятые на предприятии. Следует отметить, что непосредственное осуществление приведенного примера является весьма рискованным. Дело в том, что в этом случае не может быть предусмотрен переходной период, поскольку все три параметра имеют одинаковый тип, а это означает, что невозможно просто перекрыть рассматриваемый метод. Из этого следует, что можно переупорядочить параметры, но при этом забыть внести корректировку во внешний код, в котором вызывается метод, с учетом нового порядка расположения параметров. Таким образом, метод все еще будет выполняться успешно, но с неправильными значениями параметров, а такой дефект в программном обеспечении является очень сложным, и его нелегко обнаружить без хороших тестовых примеров. А если бы параметры относились к разным типам, то работа внешних приложений, которые не были подвергнуты рефакторингу с учетом передачи параметров в новом порядке, оканчивалась бы неудачей, поэтому поиск кода, подверженного изменениям, стал бы гораздо проще.

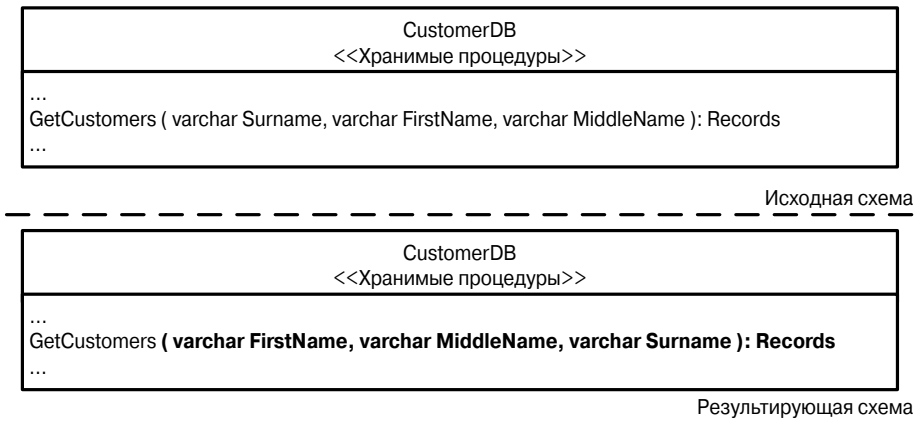


Рис. 10.6. Переупорядочение параметров метода без переходного периода

Операция рефакторинга "Замена параметра явно заданными методами"

Иногда обнаруживается такая ситуация, что с помощью одного и того же метода выполняется одно из нескольких действий в зависимости от значения переданного ему параметра. Например, на рис. 10.7 показано, что метод `GetAccountValue`, который представляет собой универсальный метод чтения, способный возвратить значение любого столбца в строке, заменяется более специализированными хранимыми процедурами `GetAccountBalance`, `GetAccountCustomerID` и `GetAccountOpeningDate`.

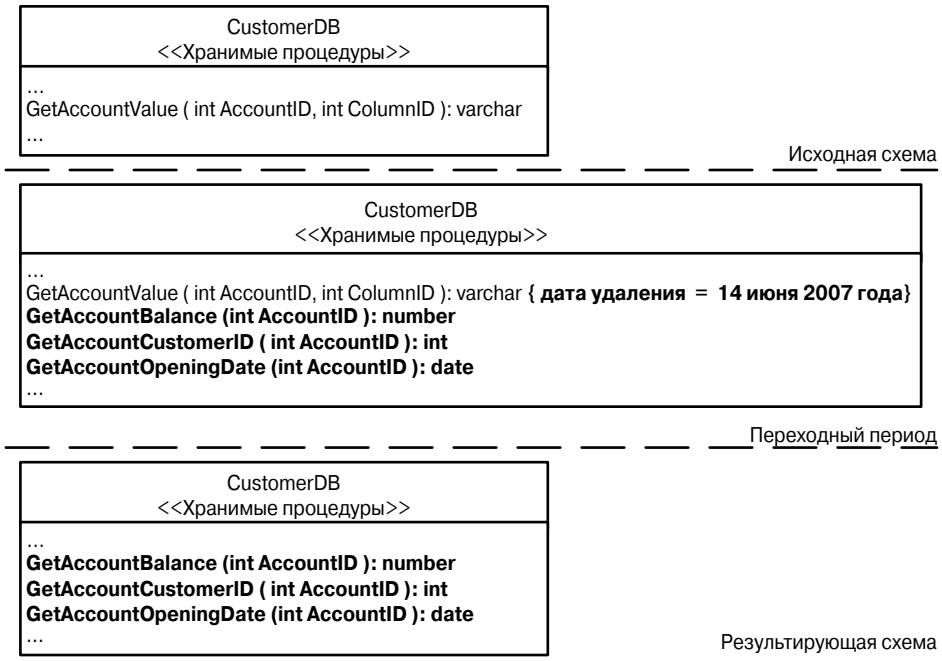


Рис. 10.7. Замена параметра явно заданными хранимыми процедурами

10.2. Операции рефакторинга внутренней организации методов

Операции рефакторинга внутренней организации методов позволяют усовершенствовать качество реализации метода без изменения его интерфейса.

Операция рефакторинга "Консолидация условного выражения"

В составе программных средств метода часто обнаруживается ряд проверок условий, которые вырабатывают один и тот же результат. Такой код следует подвергнуть рефакторингу и объединить соответствующие условные выражения, чтобы он стал более понятным, как показано в следующем примере, в котором три условных выражения объединены в одно. Эта операция рефакторинга предусматривает также применение операции "Извлечение метода" (с. 307) для создания метода, реализующего проверку условия:

```
-- Код до рефакторинга

CREATE OR REPLACE FUNCTION GetAccountAverageBalance
 ( inAccountID IN NUMBER)
 RETURN NUMBER;
AS
 averageBalance := 0;
BEGIN
 IF inAccountID = 10000 THEN
```

```
  RETURN 0;
 END IF;
 IF inAccountID = 123456 THEN
  RETURN 0;
 END IF;
 IF inAccountID = 987654 THEN
  RETURN 0;
 END IF;
 -- Код расчета среднего остатка
 RETURN averageBalance;
END;

-- Код после рефакторинга

CREATE OR REPLACE FUNCTION GetAccountAverageBalance
 ( inAccountID IN NUMBER)
 RETURN NUMBER;
AS
 averageBalance := 0;
BEGIN
 IF inAccountID = 10000 || inAccountID = 123456 || inAccountID =
987654 THEN
  RETURN 0;
 END IF;

 -- Код расчета среднего остатка
  RETURN averageBalance;
END;
```

Операция рефакторинга "Декомпозиция условного выражения"

Иногда в программе встречаются условные выражения `IF-THEN-ELSE`, которые являются сложными и поэтому затруднительными для понимания. При выполнении рассматриваемой операции рефакторинга, в частности, может быть предусмотрена замена частей условного оператора `IF`, `THEN` и `ELSE` вызовами методов, которые реализуют действия, выполняемые условным выражением. Ниже приведена первая версия кода реализации алгоритма определения применимой процентной ставки с учетом того, является ли значение `inBalance` более низким, равным или превышающим определенное пороговое значение. А во второй версии код становится проще в результате введения вызовов методов `BalanceIsSufficient`, `CalculateLowInterest` и `CalculateHighInterest`. (Соответствующие функции не показаны.)

```
-- Код до рефакторинга

CREATE OR REPLACE FUNCTION CalculateInterest
 ( inBalance IN NUMBER )
 RETURN NUMBER;
AS
 lowBalance NUMBER;
 highBalance NUMBER;
 lowInterestRate NUMBER;
 highInterestRate NUMBER;
```

```
BEGIN

 lowBalance := GetLowBalance();
 highBalance := GetHighBalance();
 lowInterestRate := GetLowInterestRate();
 highInterestRate := GetHighInterestRate();

 IF inBalance < lowBalance THEN
  RETURN 0;
 END IF

 IF inBalance >= lowBalance && inBalance <= highBalance THEN
  RETURN inBalance * lowInterestRate;
 ELSE
  RETURN inBalance * highInterestRate;
 END IF;
END;

-- Код после рефакторинга

CREATE OR REPLACE FUNCTION CalculateInterest
 ( inBalance IN NUMBER )
 RETURN NUMBER;
AS
BEGIN

 IF BalanceIsInsufficient( inBalance ) THEN
  RETURN 0;
 END IF

 IF IsLowInterestBalance( inBalance ) THEN
  RETURN CalculateLowInterest( inBalance );
 ELSE
  RETURN CalculateHighInterest( inBalance );
 END IF;
END;
```

Операция рефакторинга “Извлечение метода”

Иногда обнаруживается такая ситуация, что существующий метод содержит фрагмент кода, который может быть сгруппирован и выделен в собственный метод с именем, поясняющим его назначение. Еще одной особенностью таких фрагментов кода является также то, что они дублируются в другом месте и (или) реализуют значимые функциональные средства, поэтому применение рассматриваемой операции рефакторинга должно способствовать упрощению кода. Короткие программные модули, имеющие выразительные имена, являются более простыми для понимания и поэтому для сопровождения, кроме того, выше вероятность того, что они будут использоваться повторно. Как показано в следующем коде, первая версия упрощается путем замены программных средств определения начального остатка за текущий день вызовом хранимой функции `GetDailyBalance`. Окончательная версия этого кода характеризуется тем, что для ее максимального упрощения рассматриваемая операция рефакторинга была применена еще несколько раз. Побочным эффектом применения рассматриваемой операции рефакторинга становится то, что общедоступные храни-

мые процедуры высокого уровня выглядят как хорошо прокомментированный, очень грамотный код:

```
-- Первоначальная версия кода

CREATE OR REPLACE FUNCTION CalculateAccountInterest
 ( inAccountID IN NUMBER,
 inStart IN DATE,
 inEnd IN DATE )
 RETURN NUMBER;
AS
 medianBalance NUMBER;
 startBalance NUMBER;
 endBalance NUMBER;
 interest := 0;
BEGIN
 BEGIN
  -- Определение начального остатка
  SELECT Balance INTO startBalance
   FROM DailyEndBalance
   WHERE AccountID = inAccountID && PostingDate = inStart;
  EXCEPTION WHEN NO_DATA_FOUND THEN
   startBalance := 0;

  -- Определение конечного остатка
  SELECT Balance INTO endBalance
   FROM DailyEndBalance
   WHERE AccountID = inAccountID && PostingDate = inEnd;
  EXCEPTION WHEN NO_DATA_FOUND THEN
   endBalance := 0;
  END;
  medianBalance := ( startBalance + endBalance ) / 2;
  IF medianBalance < 0 THEN
   medianBalance := 0;
              END IF;
             IF medianBalance >= 500 THEN
   interest := medianBalance * 0.01;
  END IF;
 RETURN interest;
END;

-- Промежуточная версия кода

CREATE OR REPLACE Function CalculateAccountInterest
 ( inAccountID IN NUMBER,
 inStart IN DATE,
 inEnd IN DATE )
 RETURN NUMBER;
AS
 medianBalance NUMBER;
 startBalance NUMBER;
 endBalance NUMBER;
 interest := 0;
BEGIN
 startBalance := GetDailyEndBalance ( inAccountID, inStart );
```

```
 BEGIN
   -- Определение конечного остатка
  SELECT Balance INTO endBalance
   FROM DailyEndBalance
   WHERE AccountID = inAccountID && PostingDate = inEnd;
  EXCEPTION WHEN NO_DATA_FOUND THEN
   endBalance := 0;
 END;
 medianBalance := ( startBalance + endBalance ) / 2;
 IF medianBalance < 0 THEN
  medianBalance := 0;
 END IF;

 IF medianBalance >= 500 THEN
  interest := medianBalance * 0.01;
 END IF;
 RETURN interest;
END;

CREATE OR REPLACE Function GetDailyBalance
 ( inAccountID IN NUMBER,
 inDate IN DATE )
 RETURN NUMBER;
AS
 endbalance NUMBER;
BEGIN
 BEGIN
   SELECT Balance INTO endBalance
   FROM DailyEndBalance
   WHERE AccountID = inAccountID && PostingDate = inDate;
  EXCEPTION WHEN NO_DATA_FOUND THEN
   endBalance := 0;
 END;
 RETURN endBalance;
END;

-- Окончательная версия кода

CREATE OR REPLACE FUNCTION CalculateAccountInterest
( inAccountID IN NUMBER,
 inStart IN DATE,
 inEnd IN DATE )
 RETURN NUMBER;
AS
 medianBalance NUMBER;
 startBalance NUMBER;
 endBalance NUMBER;
BEGIN
 startBalance := GetDailyEndBalance ( inAccountID, inStart );
 endBalance:= GetDailyEndBalance ( inAccountID, inEnd );
 medianBalance := CalculateMedianBalance ( startBalance, endBalance );
 RETURN CalculateInterest ( medianBalance );
END;

--  Здесь должен находиться код GetDailyEndBalance,
CalculateMedianBalance и CalculateInterest
```

Операция рефакторинга “Введение переменной”

Иногда обнаруживается, что код метода содержит сложное выражение, которое является трудным для чтения. В таком случае в код могут быть введены переменные, имеющие наглядные имена, которые используются для присваивания им значений частей выражения, а затем включаются в окончательное выражение, заменяющее исходное выражение, в результате чего метод становится более простым для понимания. Рассматриваемая операция рефакторинга является эквивалентом операции “Введение пояснительной переменной”, предложенной Мартином Фаулером [Fowler, 1999]. Авторы обсудили с Мартином предлагаемую операцию рефакторинга, после чего он предложил предусмотреть для этой операции более простое имя; дело в том, что Мартин понял, что должен был с самого начала назвать эту операцию рефакторинга как “Введение переменной”, к тому же большинство поставщиков инструментальных средств также используют более простое имя:

```
-- Код до рефакторинга

CREATE OR REPLACE FUNCTION DetermineAccountStatus
 ( inAccountID IN NUMBER,
  inStart IN DATE,
  inEnd IN DATE )
 RETURN VARCHAR;
AS
      lastAccessedDate DATE;
BEGIN
 -- Код вычисления значения lastAccessDate

 IF ( inDate < lastAccessDate && outdate > lastAccessDate )
  && ( inAccountID > 10000 )
  && ( inAccountID != 123456 && inAcountID != 987654) THEN
  -- Выполнение некоторых действий
 END IF;
 -- Выполнение других действий
END;

-- Код после рефакторинга

CREATE OR REPLACE FUNCTION DetermineAccountStatus
 ( inAccountID IN NUMBER,
  inStart IN DATE,
  inEnd IN DATE )
 RETURN VARCHAR;
AS
 lastAccessedDate DATE;
 isBetweenDates BOOLEAN;
 isValidAccountID BOOLEAN;
 isNotTestAccount BOOLEAN
BEGIN
 -- Код вычисления значения lastAccessDate
 isBetweenDates := inDate < lastAccessDate && outdate >
lastAccessDate;
 isValidAccountID := inAccountID > 100000;
 isNotTestAccount := inAccountID != 123456 && inAcountID != 987654;
```

```
 IF isBetweenDates && isValidAccountID && isNotTestAccount THEN
  -- Выполнение некоторых действий
 END IF;
 -- Выполнение других действий
END;
```

Операция рефакторинга "Удаление флажка управления"

Предположим, что в программе предусмотрена переменная, действующая как флажок управления, позволяющий выйти из управляющей конструкции, такой как цикл. Рассматриваемая операция рефакторинга позволяет упростить код путем смены назначения флажка управления в результате исследования выражения, которое задано в управляющей конструкции. Как показано в следующем примере, после удаления программных средств, связанных с применением переменной `controlFlag`, код упрощается:

```
-- Код до рефакторинга

DECLARE
 controlFlag := 0;
 anotherVariable := 0;
BEGIN
 WHILE controlFlag = 0 LOOP
  -- Выполнение некоторых действий
  IF anotherVariable > 20 THEN
   controlFlag = 1;
  ELSE
   -- Выполнение еще каких-то действий
  END IF;
 END LOOP;
END;

-- Код после рефакторинга

DECLARE
 anotherVariable := 0;
BEGIN
 WHILE anotherVariable <= 20 LOOP
  -- Выполнение некоторых действий
  -- Выполнение еще каких-то действий
 END LOOP;
END;
```

Операция рефакторинга "Удаление посредника"

Иногда обнаруживается такая ситуация, что некоторый метод служит лишь в качестве передаточного звена, или посредника, для других методов. Эта ситуация может возникнуть, если какая-то хранимая процедура была переименована, а затем введена другая хранимая процедура с ее первоначальным именем, которое просто вызывает прежнюю хранимую процедуру по новому имени. Еще одна возможность возникает, если обнаруживается, что имеются две хранимые процедуры, которые выполняют одни и те же действия; в таком случае, вполне очевидно, одна из них корректируется так, чтобы в ней выполнялся вызов другой. Так или иначе, при обнаружении кода, подобного приведенному

в следующем примере, необходимо подвергнуть рефакторингу все участки кода, в которых вызывается AProcedure, чтобы в дальнейшем в них вызывалась AnotherProcedure (что дает возможность удалить AProcedure):

```
CREATE OR REPLACE PROCEDURE AProcedure
 parameter1 IN NUMBER;
 ...
 parameterN IN VARCHAR;
AS
BEGIN
 EXECUTE AnotherProcedure ( parameter1, ..., parameterN );
END;
```

Операция рефакторинга "Переименование параметра"

Часто обнаруживается такая ситуация, что существующий параметр имеет имя, затруднительное для понимания. Это может быть связано с тем, что имя параметра в свое время имело смысл, но в дальнейшем назначение самого параметра изменилось или же просто этот параметр с самого начала был назван неправильно. Рассматриваемая операция рефакторинга является довольно простой для реализации; достаточно просто изменить имя параметра в первоначальном варианте исходного кода.

Операция рефакторинга "Замена подстановки литерала поиском в таблице"

Иногда обнаруживается, что в код метода включено с применением программных конструкций литеральное числовое значение, которое имеет определенный смысл, но сопровождение этого значения является затруднительным. Лучший подход состоит в том, чтобы сохранить подобное значение в таблице, а затем осуществлять его выборку из таблицы по мере необходимости. (Эта таблица может быть кэширована в целях повышения производительности.) В следующем примере показано, что в новой версии кода предусмотрена выборка значения минимального остатка на счете, равного 500 долл., из единственной строки таблицы, называемой CorporateBusinessConstants, которая предназначена для хранения подобных значений. (Может быть также предусмотрено использование многострочной таблицы с ограниченным количеством столбцов, по одному для каждого типа, в которых должны храниться такие данные.) Рассматриваемый код может быть дополнительно усовершенствован путем повторного применения этой операции рефакторинга для получения значения, соответствующего процентной ставке. Кроме того, для объединения программных средств, предназначенных для получения значений из поисковой таблицы, можно применить операцию "Извлечение метода" (с. 307). Рассматриваемая операция рефакторинга представляет собой версию операции рефакторинга кода "Замена магического числа символической константой" ([Fowler 1999]), относящуюся к базе данных:

```
-- Первоначальная версия кода

CREATE OR REPLACE FUNCTION CalculateInterest
 ( inBalance IN NUMBER )
 RETURN NUMBER;
AS
```

```
 interest := 0;
BEGIN
 IF inBalance >= 500 THEN
  interest := medianBalance * 0.01;
 END IF;
 RETURN interest;
END;

-- Промежуточная версия кода

CREATE OR REPLACE FUNCTION CalculateInterest
 ( inBalance IN NUMBER )
 RETURN NUMBER;
AS
 interest := 0;
 minimumBalance NUMBER;
BEGIN
 BEGIN
  SELECT MinimumBalanceForInterest INTO minimumBalance
   FROM CorporateBusinessConstants
   WHERE RowNumber = 1;
  EXCEPTION WHEN NO_DATA_FOUND THEN
   minimumBalance := 0;
 END;
 IF inBalance >= minimumBalance THEN
  interest := medianBalance * 0.01;
 END IF;
 RETURN interest;
END;

-- Окончательная версия кода

CREATE OR REPLACE FUNCTION CalculateInterest
 ( inBalance IN NUMBER )
 RETURN NUMBER;
AS
 interest := 0;
 minimumBalance NUMBER;
 interestRate NUMBER;
BEGIN

 minimumBalance := GetMinimumBalance();
 interestRate := GetInterestRate();

 IF inBalance >= minimumBalance THEN
  interest := medianBalance * interestRate;
 END IF;
 RETURN interest;
END;
```

Операция рефакторинга "Замена вложенного условного выражения защитными конструкциями"

Иногда операторы с вложенными условными выражениями становятся сложными для понимания. В следующем примере показано, что в соответствии с рассматриваемой операцией рефакторинга вложенные операторы `IF` были заменены последовательностью отдельных операторов `IF` в целях повышения удобства кода для чтения:

```
-- Код до рефакторинга

BEGIN
 IF condition1 THEN
  -- Выполнение некоторых действий по условию 1
 ELSE
  IF condition2 THEN
   -- Выполнение некоторых действий по условию 2
  ELSE
   IF condition3 THEN
    -- Выполнение некоторых действий по условию 3
   END IF;
  END IF;
 END IF;
END;

-- Код после рефакторинга

BEGIN
 IF condition1 THEN
  -- Выполнение некоторых действий по условию 1
  RETURN;
 END IF;
 IF condition2 THEN
  -- Выполнение некоторых действий по условию 2
  RETURN;
 END IF;
 IF condition3 THEN
  -- Выполнение некоторых действий по условию 3
  RETURN;
 END IF;
END;
```

Операция рефакторинга "Разбиение временной переменной"

В некоторых случаях одна и та же временная переменная используется в методе для одной или нескольких целей. В результате этого чаще всего исключается возможность присвоить такой переменной осмысленное имя или, возможно, имя временной переменной соответствует одному назначению, но не другому. Решение состоит в том, чтобы ввести отдельную временную переменную для каждого назначения, как показано в следующем коде, в котором временная переменная `aTemporaryVariable` используется для хранения значений, представленных в британской системе единиц, которые преобразуются в метрическую систему единиц:

```
-- Код до рефакторинга

DECLARE
 aTemporaryVariable := 0;
 farenheitTemperature := 0;
 lengthInInches := 0;
BEGIN
 -- Получение значения farenheitTemperature
 aTemporaryVariable := (farenheitTemperature - 32 ) * 5 / 9;
 -- Выполнение некоторых действий
 -- Получение значения lengthInInches
 aTemporaryVariable := lengthInInches * 2.54;
 -- Выполнение некоторых действий
END;

-- Код после рефакторинга

DECLARE
 celciusTemperature := 0;
 farenheitTemperature := 0;
 lenghtInCentimeters := 0;
 lengthInInches := 0;
BEGIN
 -- Получение значения farenheitTemperature
 celciusTemperature := (farenheitTemperature - 32 ) * 5 / 9;
 -- Выполнение некоторых действий
 -- Получение значения lengthInInches
 lenghtInCentimeters := lengthInInches * 2.54;
 -- Выполнение некоторых действий
END;
```

Операция рефакторинга “Подстановка алгоритма”

Иногда обнаруживается, что есть более наглядный способ оформления в виде кода программных средств реализации алгоритма, который в настоящее время реализован в виде конкретного метода; в этом случае следует обязательно воспользоваться открывающейся возможностью. Если же необходимо внести изменения в существующий алгоритм, то чаще всего обнаруживается, что вначале лучше его упростить, а затем обновить. Как правило, задача повторной реализации сложного алгоритма является весьма затруднительной, поэтому вначале следует его упростить с помощью других операций рефакторинга, таких как “Извлечение метода” (с. 307), и лишь затем применять операцию “Подстановка алгоритма”.

Глава 11

Преобразования

Преобразования представляют собой изменения, в результате внесения которых изменяется семантика схемы базы данных путем добавления к схеме новых средств. В настоящей главе рассматриваются перечисленные ниже преобразования.

- Преобразование “Вставка данных”.
- Преобразование “Введение нового столбца”.
- Преобразование “Введение новой таблицы”.
- Преобразование “Введение представления”.
- Преобразование “Обновление данных”.

Преобразование “Вставка данных”

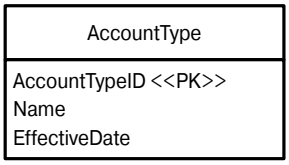
Это преобразование позволяет вставить данные в существующую таблицу (рис. 11.1).

Обоснование

Необходимость в применении преобразования “Вставка данных”, как правило, возникает в результате структурных изменений в проекте таблицы. Чаще всего преобразование “Вставка данных” приходится применять после осуществления описанных ниже действий.

- **Реорганизация таблицы.** После проведения операции рефакторинга “Переименование таблицы” (с. 148), “Слияние таблиц” (с. 133), “Разбиение таблицы” (с. 177) или “Удаление таблицы” (с. 117) может потребоваться выполнить преобразование “Вставка данных” для реорганизации данных в существующих таблицах.
- **Обеспечение доступа к статическим поисковым данным.** Статические поисковые данные требуются для любых приложений; таковыми, например, являются таблицы со списками штатов/провинций (таких как Иллинойс и Онтарио), в которых проводятся деловые операции, со списками типов адресов (к ним относятся, допустим, домашний, служебный адрес и адрес на время отпуска), а также со списками типов счетов (например, расчетный, сберегательный и инвестиционный). Если в приложении не предусмотрены административные окна редактирования для сопровождения подобных списков, то может потребоваться вставить эти данные вручную.

Схема



Значения данных до рефакторинга

AccountType		
AccountTypeID	Name	EffectiveDate
1	Checking	Dec 6 2005
2	Saving	Nov 12 2005
3	Private	Jan 7 2006
4	Money Market	Jun 4 2006
5	Credit	Mar 12 2006

Значения данных после рефакторинга

AccountType		
AccountTypeID	Name	EffectiveDate
1	Checking	Dec 6 2005
2	Saving	Nov 12 2005
3	Private	Jan 7 2006
4	Money Market	Jun 4 2006
5	Credit	Mar 12 2006
6	**Brokerage**	**Feb 1 2007**

Рис. 11.1. Вставка нового значения `AccountType`

- **Создание испытательных данных.** В процессе разработки приложений приходится использовать известные значения данных, вставляемые в базу (базы) данных, применяемую при разработке, для обеспечения возможности проверки приложений.

Потенциальные преимущества и недостатки

Задача вставки новых данных в таблицы может оказаться сложной, особенно если требуется вставить справочные данные, на которые ссылается одна или несколько других таблиц. Например, предположим, что таблица `Address` ссылается на таблицу `State`, в которую вставляются новые данные о штатах, провинциях или территориях. Данные, которые вставляются в таблицу `State`, должны содержать допустимые значения, входящие в состав адресов, представленных в таблице `Address`.

Процедура обновления схемы

Для обновления схемы базы данных необходимо выполнить описанные ниже действия.

1. **Определить данные, подлежащие вставке.** К этому относится выявление всех зависимостей. Кроме того, необходимо выяснить, должна ли быть удалена исходная строка, если рассматриваемая операция связана с перемещением данных из другой таблицы. К тому же необходимо определить, являются ли вставляемые данные новыми, и в случае положительного ответа на этот вопрос убедиться в том, что эти данные утверждены лицом (лицами), заинтересованным в разработке текущего проекта.
2. **Определить место назначения данных.** Для этого следует определить, в какую таблицу должны быть вставлены данные.
3. **Определить источник данных.** К этому относится поиск ответа на вопрос о том, поступают ли данные из другой таблицы или должны быть вставлены вручную (например, должен ли быть написан сценарий для создания необходимых данных).
4. **Уточнить необходимость в преобразовании.** При этом следует определить, требуется ли преобразовывать данные перед вставкой в целевую таблицу. Например, может потребоваться подготовить список метрических единиц измерения, которые должны быть преобразованы в британские единицы измерения перед вставкой в поисковую таблицу.

Процедура переноса данных

Если требуется вставить небольшой объем данных, то, скорее всего, будет обнаружено, что достаточно предусмотреть использование простого сценария на языке SQL, с помощью которого будут вставлены исходные данные в целевое местоположение. Если же объем данных достаточно велик, то возникает необходимость в использовании более сложного подхода, например, основанного на применении утилит баз данных, таких как SQLLDR для СУБД Oracle, или массового загрузчика (поскольку в противном случае для вставки данных потребуется много времени).

На рис. 11.1 показано, как вставить в таблицу `AccountType` новую строку, представляющую брокерские счета. Вставленное в результате значение в таблице `AccountType` поддерживает новое функциональное средство, которое должно быть вначале проверено, а затем включено в производственные приложения. В следующем коде приведены операторы DML, необходимые для вставки данных в таблицу `AccountType`:

```
INSERT INTO AccountType
(AccountTypeId,Name,EffectiveDate)
VALUES
(6,'Brokerage','Feb 1 2007');
```

Процедура обновления программ доступа

Если операция рефакторинга “Вставка данных” применяется в связи с проведением какой-то другой операции рефакторинга базы данных, то с учетом результатов ее применения необходимо также обновить все приложения. Но если преобразование “Вставка данных” служит для добавления данных в существующую схему, то, возможно, потребу-

ется обновить код внешнего приложения. В частности, может потребоваться обновить конструкцию WHERE. Например, на рис. 11.1 показано, что в таблицу AccountType вставлено значение Brokerage. Поэтому может возникнуть необходимость обновить операторы SELECT, для того чтобы в них не считывалось это значение, если должна быть обеспечена работа только со стандартными банковскими счетами (со всеми счетами, кроме брокерских), как показано в приведенном ниже коде. Может также потребоваться проведение операции рефакторинга "Введение представления" (с. 325) для создания конкретного представления, используемого приложением, для того чтобы это представление возвращало лишь подмножество данных:

```
// Код до рефакторинга
stmt.prepare(
  "SELECT * FROM AccountType " +
    "WHERE AccountTypeId NOT IN (?,?)");
stmt.setLong(1,PRIVATEACCOUNT.getId);
stmt.setLong(2,MONEYMARKETACCOUNT.getId);
stmt.execute();
ResultSet standardAccountTypes = stmt.executeQuery();

// Код после рефакторинга
stmt.prepare(
  "SELECT * FROM AccountType " +
    "WHERE AccountTypeId NOT IN (?,?,?)");
stmt.setLong(1,PRIVATEACCOUNT.getId);
stmt.setLong(2,MONEYMARKETACCOUNT.getId);
stmt.setLong(3,BROKERAGE.getId);
stmt.execute();
ResultSet standardAccountTypes = stmt.executeQuery();
```

Аналогичным образом, может потребоваться обновить исходный код, в котором проверяется допустимость значений атрибутов данных. Например, предположим, что в составе программных средств приложения имеется код, в котором определены премиальные счета, которые относятся к типу Private или Money Market. А теперь к этому списку необходимо добавить счета типа Brokerage, как показано в следующем коде:

```
// Код до рефакторинга
public enum PremiumAccountType {
  PRIVATEACCOUNT(new Long(3)),
  MONEYMARKET(new Long(4));

private Long id;
public Long getId() {
  return id;
}

PremiumAccountType(Long value) {
  this.id = value;
}

public static Boolean
  isPremiumAccountType(Long idToFind) {
    for (PremiumAccountType premiumAccountType :
          PremiumAccountType.values()) {
```

```
      if (premiumAccountType.id.equals(idToFind))
         return Boolean.TRUE;
       }
        Return Boolean.FALSE
}

// Код после рефакторинга
public enum PremiumAccountType {
  PRIVATEACCOUNT(new Long(3)),
  MONEYMARKET(new Long(4)),
  BROKERAGE(new Long(6));

private Long id;
public Long getId() {
  return id;
}

PremiumAccountType(Long value) {
  this.id = value;
}

public static Boolean
  isPremiumAccountType(Long idToFind) {
    for (PremiumAccountType premiumAccountType :
          PremiumAccountType.values()) {
      if (premiumAccountType.id.equals(idToFind))
         return Boolean.TRUE;
       }
        Return Boolean.FALSE
}
```

Преобразование “Введение нового столбца”

Это преобразование позволяет ввести новый столбец в существующую таблицу (рис. 11.2).

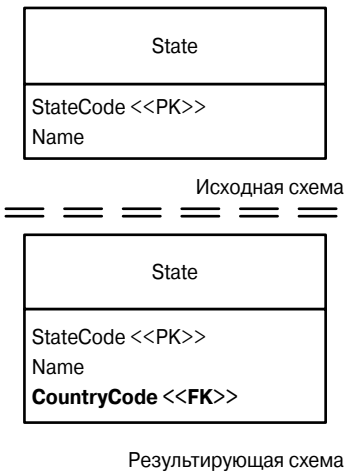


Рис. 11.2. Ввод в таблицу столбца `State.CountryCode`

Обоснование

Необходимость в применении преобразования "Введение нового столбца" может быть обусловлена несколькими описанными ниже причинами.

- **Ввод в действие нового атрибута.** Необходимость добавления нового столбца в базу данных может быть связана с выдвижением дополнительных требований к базе данных.
- **Проведение промежуточного шага некоторой операции рефакторинга.** Промежуточный шаг, в котором в существующую таблицу вводится новый столбец, должен быть предусмотрен во многих операциях рефакторинга базы данных, таких как "Перемещение столбца" (с. 139) и "Переименование столбца" (с. 145).

Потенциальные преимущества и недостатки

Необходимо обеспечить, чтобы вводимый столбец больше не существовал где-либо в другом месте, поскольку в противном случае возникает риск обострения проблем ссылочной целостности из-за увеличения избыточности данных.

Процедура обновления схемы

Как показано на рис. 11.2, для обновления схемы базы данных необходимо ввести дополнительный столбец `CountryCode` с помощью конструкции `ADD` команды `ALTER TABLE` языка SQL. Столбец `State.CountryCode` представляет собой ссылку на внешний ключ в таблице `Country` (не показана), позволяющую приложению (приложениям) отслеживать данные об административных единицах государств. В следующем коде приведены операторы DDL, предназначенные для введения столбца `State.CountryCode`. Для создания в связи с этим ограничения ссылочной целостности применяется операция рефакторинга "Добавление ограничения внешнего ключа" (с. 229):

```
ALTER TABLE State ADD Country Code VARCHAR2(3) NULL;
```

Процедура переноса данных

Безусловно, при проведении этого преобразования основная сложность заключается не в переносе данных как таковом, а в заполнении столбца значениями данных после добавления его в таблицу. Для этого необходимо выполнить описанные ниже действия.

1. Провести работу с лицами, заинтересованными в создании проекта, в целях выявления соответствующих значений.
2. Ввести новые значения в столбец вручную или написать сценарий для автоматического заполнения столбца (или же воспользоваться двумя этими подходами).
3. В зависимости от конкретных условий может потребоваться предусмотреть возможность применения к этому новому столбцу операций рефакторинга "Введение заданного по умолчанию значения" (с. 213), "Уничтожение столбца, не допускающего NULL-значений" (с. 205) или "Преобразование столбца в недопускающий NULL-значения" (с. 216).

В следующем коде приведены операторы DML, предназначенные для заполнения столбца `State.CountryCode` путем ввода начального значения `'USA'`:

```
UPDATE State SET CountryCode = 'USA'
  WHERE CountryCode IS NULL;
```

Процедура обновления программ доступа

Процедура обновления является простой, поскольку для ввода в действие нового столбца достаточно лишь предусмотреть его применение в приложении (приложениях). В следующем коде показан пример того, какие изменения потребуется внести в метаданные Hibernate-отображения `OR` в связи с добавлением столбца:

```
// Отображение до рефакторинга
<hibernate-mapping>
<class name="State" table="STATE">
  <id name="id" column="StateCode"></id>
  <property name="name"/>
</class>
</hibernate-mapping>

// Отображение после рефакторинга
<hibernate-mapping>
<class name="State" table="STATE">
  <id name="id" column="StateCode"></id>
  <property name="name"/>
  <many-to-one name="country"
               class="Country"
               column="COUNTRYCODE"/>
</class>
</hibernate-mapping>
```

Преобразование “Введение новой таблицы”

Это преобразование позволяет ввести новую таблицу в существующую базу данных (рис. 11.3).

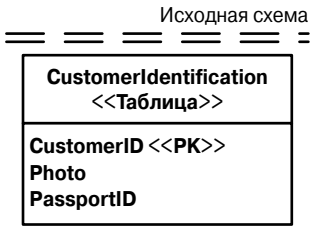


Рис. 11.3. Создание таблицы `CustomerIdentification`

Обоснование

Необходимость в применении преобразования "Введение новой таблицы" может быть обусловлена несколькими описанными ниже причинами.

- **Введение новых атрибутов.** Необходимость в добавлении новой таблицы к базе данных может быть обусловлена дополнительными требованиями.
- **Осуществление промежуточного шага другой операции рефакторинга.** Введение новой таблицы вместо существующей представляет собой один из шагов многих операций рефакторинга базы данных, таких как "Разбиение таблицы" (с. 177) и "Переименование таблицы" (с. 148).
- **Введение нового официально утвержденного источника данных.** В процессе эксплуатации весьма часто обнаруживается, что в нескольких таблицах хранится аналогичная информация. Например, может оказаться, что информация о клиентах поступает из нескольких источников, причем эти источники часто не синхронизированы друг с другом, а иногда даже содержат противоречивые данные. При таких обстоятельствах необходимо провести операцию рефакторинга "Использование официально заданного источника данных" (с. 292), а затем по истечении определенного времени применить операцию "Удаление таблицы" (с. 117) к исходным таблицам.
- **Необходимость в резервном копировании данных.** При реализации некоторых операций рефакторинга, таких как "Удаление таблицы" (с. 117) или "Слияние таблиц" (с. 133), может потребоваться создать таблицу для хранения табличных данных в целях промежуточного использования или резервного копирования.

Потенциальные преимущества и недостатки

Основным требованием, связанным с выполнением рассматриваемого преобразования, является обеспечение того, чтобы создаваемая таблица не существовала уже в базе данных под другим именем. Кроме того, часто возникает такая ситуация, что именно та таблица, которая требуется, не существует, но имеется нечто близкое к ней; в этом случае лучше подвергнуть рефакторингу существующую таблицу, чем добавить новую, содержащую избыточную информацию.

Процедура обновления схемы

Как показано на рис. 11.3, для обновления схемы базы данных необходимо ввести новую таблицу `CustomerIdentification` с помощью команды `CREATE TABLE` языка SQL. В следующем коде приведены операторы DDL, предназначенные для создания таблицы `CustomerIdentification`. Столбец `CustomerIdentification.CustomerID` представляет собой ссылку на внешний ключ в таблице `Customer` (не показана), позволяющую приложению (приложениям) отслеживать различные способы идентификации отдельных клиентов:

```
CREATE TABLE CustomerIdentification(
  CustomerID NUMBER NOT NULL,
  Photo      BLOB,
  PassportID NUMBER,
  CONSTRAINT PK_CustomerIdentification
    PRIMARY KEY (CustomerID)
  );
```

Процедура переноса данных

Безусловно, при проведении этого преобразования основная сложность заключается не в переносе данных как таковом, а в заполнении таблицы значениями данных после добавления ее в базу данных. Для этого необходимо выполнить описанные ниже действия.

1. Провести работу с лицами, заинтересованными в создании проекта, в целях выявления соответствующих значений данных.
2. Ввести новые значения в таблицу вручную или написать сценарий для автоматического заполнения таблицы столбца (или же прибегнуть к сочетанию двух этих подходов).
3. Рассмотреть возможность применения операции рефакторинга “Вставка данных” (с. 317).

Процедура обновления программ доступа

В идеальном случае при использовании рассматриваемого преобразования процедура обновления схемы является несложной, поскольку для работы с новой таблицей достаточно обеспечить ее применение в приложении (приложениях). Но если новая таблица вводится в качестве замены нескольких других таблиц, то часто обнаруживается, что новая таблица имеет немного другую схему, а что еще хуже, реализует немного другую семантику данных по сравнению с существующими таблицами. Если дело обстоит таким образом, то возникает необходимость подвергнуть рефакторингу программы доступа для обеспечения работы с новой версией.

Преобразование “Введение представления”

Это преобразование позволяет создать новое представление, основанное на таблицах, существующих в базе данных (рис. 11.4).

Обоснование

Необходимость в применении преобразования “Введение представления” может быть обусловлена несколькими приведенными ниже причинами.

- **Подготовка данных для отчетов.** Для многих отчетов требуются итоговые данные, которые можно подготовить с помощью определения представления.
- **Исключение избыточных операций чтения.** Иногда возникает такая ситуация, что в нескольких внешних программах или в нескольких хранимых процедурах (предназначенных для этих программ) применяется один и тот же запрос на выполнение операции выборки. Все подобные запросы могут быть заменены общедоступной таблицей, допускающей только чтение, или представлением.
- **Защита данных.** Представление может использоваться как средство, с помощью которого конечные пользователи получают права доступа к данным для чтения, но без прав на обновление.

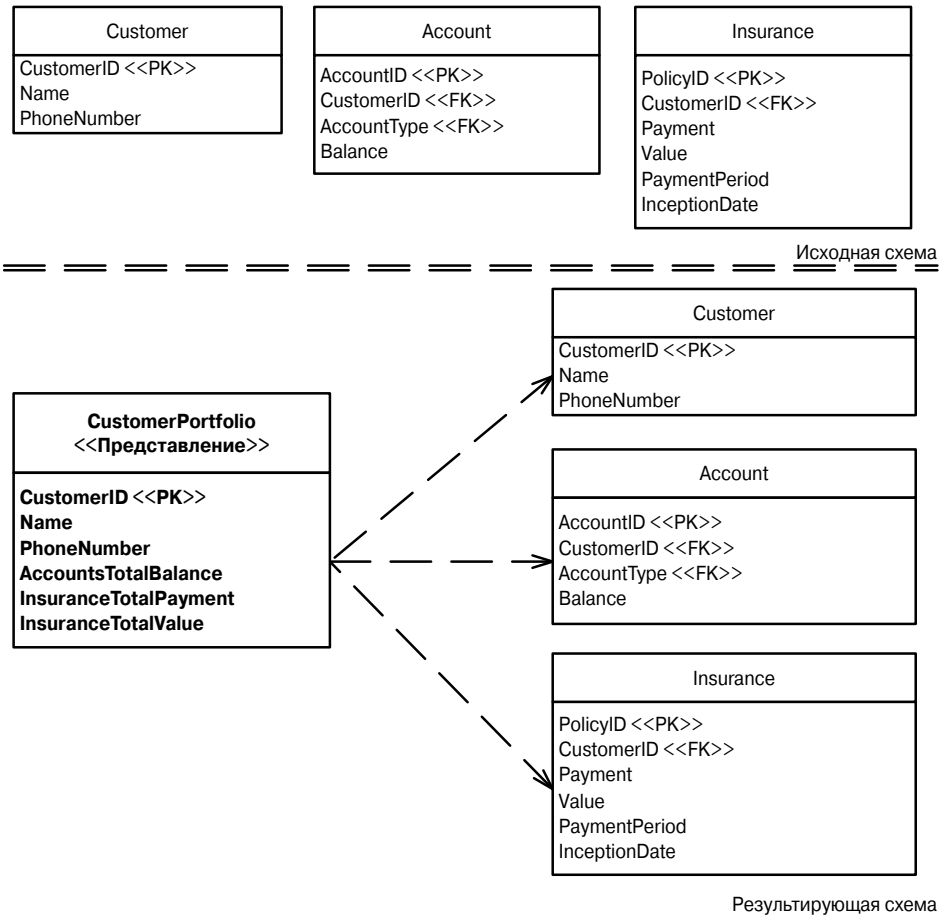


Рис. 11.4. Введение представления `CustomerPortfolio`

- **Инкапсуляция средств доступа к таблице.** Некоторые организации предпочитают применять инкапсуляцию средств доступа к таблице путем определения вместо исходных таблиц обновляемых представлений, к которым получают доступ внешние программы. Благодаря этому организации получают возможность легко осуществлять такие операции рефакторинга базы данных, как "Переименование столбца" (с. 145) или "Переименование таблицы" (с. 148), исключая при этом вероятность нарушения работы внешних приложений, поскольку представления создают дополнительный уровень инкапсуляции между таблицами и приложением.
- **Сокращение объема дублирования кода SQL.** Если в приложении применяются сложные запросы SQL, то часто обнаруживается, что отдельные фрагменты такого кода SQL дублируются во многих местах. Если дело обстоит таким образом, то необходимо ввести в действие представления для реализации в них дублирующегося кода SQL, как показано в разделе с примерами.

Потенциальные преимущества и недостатки

При вводе в действие представления необходимо решить две основные задачи. Во-первых, для конечных пользователей может оказаться неприемлемой производительность соединений, определяемых с помощью представления, в связи с чем может потребоваться применить другой подход, например реализовать операцию рефакторинга “Введение таблицы только для чтения” (с. 273). Во-вторых, в результате введения нового представления связность объектов в схеме базы данных увеличивается, поскольку, как показано на рис. 11.4, определение представления зависит от определения (определений) схемы таблицы.

Процедура обновления схемы

Для обновления схемы базы данных при выполнении операции “Введение представления” необходимо ввести представление с помощью команды `CREATE VIEW`. На рис. 11.4 показан пример, в котором применяется представление `CustomerPortfolio`, основанное на таблицах `Customer`, `Account` и `Insurance`, для суммирования данных о сделках, заключенных с каждым клиентом. В результате ввода в действие такого представления конечные пользователи получают более удобный способ выполнения произвольных запросов. В следующем коде приведены операторы DDL, предназначенные для создания представления `CustomerPortfolio`:

```
CREATE VIEW CustomerPortfolio (
  CustomerID NUMBER NOT NULL,
  Name VARCHAR(40),
  PhoneNumber VARCHAR2(40),
  AccountsTotalBalance NUMBER,
  InsuranceTotalPayment NUMBER,
  InsuranceTotalValue NUMBER,
) AS SELECT
  Customer.CustomerID,
  Customer.Name,
  Customer.PhoneNumber,
  SUM(Account.Balance),
  SUM(Insurance.Payment),
  SUM(Insurance.Value)
  FROM
    Customer,Account,Insurance
  WHERE
    Customer.CustomerID=Account.CustomerID
    AND Customer.CustomerID=Insurance.CustomerID
;
```

Процедура переноса данных

При выполнении этой операции рефакторинга базы данных данные, подлежащие переносу, отсутствуют.

Процедура обновления программ доступа

Применяемая процедура обновления зависит от ситуации. Если рассматриваемое представление вводится для замены широко применяемого в программах доступа кода SQL выборки данных, то следует должным образом подвергнуть этот код рефакторингу, чтобы в нем использовалось новое представление. А если представление вводится для обеспечения формирования отчетов, то должны быть написаны новые отчеты, для того чтобы в них использовалось это представление. Для программ, получающих доступ к представлению, такое представление выглядит как таблица, допускающая только чтение.

Если в приложении имеется дублирующийся код SQL, то вероятность программных ошибок возрастает, поскольку после внесения изменений в код SQL в одном месте необходимо внести аналогичные изменения во все дублирующиеся фрагменты. Например, рассмотрим следующие экземпляры прикладного кода SQL:

```
SELECT Customer.CustomerID, SUM(Insurance.Payment),
SUM(Insurance.Value)
FROM Customer, Insurance
WHERE
  Customer.CustomerID=Insurance.CustomerID
  AND Customer.Status = 'ACTIVE'
  AND Customer.InBusinessSince <= TodaysDateLastYear
GROUP BY Customer.CustomerID
;

SELECT Customer.CustomerID, SUM(Account.Balace)
FROM Customer, Account
WHERE
  Customer.CustomerID=Account.CustomerID
  AND Customer.Status = 'ACTIVE'
  AND Customer.InBusinessSince <= TodaysDateLastYear
GROUP BY Customer.CustomerID
```

Как показывают два приведенных выше примера кода, та часть, в которой осуществляется выборка клиента с атрибутом `ACTIVE`, является дублирующейся. Можно создать представление, в которое вынесен этот дублирующийся код SQL, после чего предусмотреть использование в соответствующих фрагментах SQL данного представления, что позволяет избежать необходимости вносить изменения во многих местах после того, как код SQL, предназначенный для выборки данных о клиентах с атрибутом `ACTIVE`, изменится:

```
CREATE OR REPLACE VIEW ActiveCustomer
SELECT Customer.CustomerID
FROM Customer
WHERE
  Customer.Status = 'ACTIVE'
  AND Customer.InBusinessSince <= TodaysDateLastYear
;

SELECT ActiveCustomer.CustomerID, SUM(Insurance.Payment),
SUM(Insurance.Value)
FROM ActiveCustomer, Insurance
WHERE ActiveCustomer.CustomerID=Insurance.CustomerID
GROUP BY ActiveCustomer.CustomerID
```

```
;

SELECT ActiveCustomer.CustomerID, SUM(Account.Balace)
FROM ActiveCustomer, Account
WHERE ActiveCustomer.CustomerID=Account.CustomerID
GROUP BY ActiveCustomer.CustomerID
;
```

Преобразование “Обновление данных”

Это преобразование позволяет обновить данные в существующей таблице (рис. 11.5).

Схема

AccountType
AccountTypeID <<PK>> Name EffectiveDate

Значения данных до рефакторинга

AccountType		
AccountTypeID	Name	EffectiveDate
1	Checking	Dec 6 2005
2	Saving	Nov 12 2005
3	Private	Jan 7 2006
4	Money Market	Jun 4 2006
5	Credit	Mar 12 2006

Значения данных после рефакторинга

AccountType		
AccountTypeID	Name	EffectiveDate
1	Checking	Sep 29 2006
2	Saving	Nov 12 2005
3	**Private Banking**	**Sep 29 2006**
4	Money Market	Jun 4 2006
5	Credit	Mar 12 2006

Рис. 11.5. Обновление таблицы `AccountType` путем ввода новых обозначений типов

Обоснование

Необходимость в применении преобразования "Обновление данных" может быть обусловлена проведением описанных ниже действий.

- **Реорганизация таблицы.** После применения операций рефакторинга "Переименование таблицы" (с. 148), "Переименование столбца" (с. 145), "Перемещение столбца" (с. 139), "Разбиение таблицы" (с. 177), "Разбиение столбца" (с. 172), "Слияние таблиц" (с. 133) или "Слияние столбцов" (с. 130) может потребоваться использовать преобразование "Обновление данных" для реорганизации данных в существующих таблицах.
- **Предоставление до сих пор не существовавших данных.** После проведения таких преобразований, как "Введение нового столбца" (с. 321), может потребоваться предоставить данные для вновь введенного столбца в существующих производственных базах данных. Например, после того как в таблицу `Address` будет добавлен столбец `Country`, необходимо заполнить его соответствующими значениями.
- **Изменение справочных данных.** После изменения бизнес-правил возникает необходимость ввести изменения в некоторые справочные/поисковые данные. Например, допустим, что в связи с изменением деловой терминологии необходимо откорректировать значение `'Private'` в столбце `AccountType.Name`, заменив его значением `'Private Banking'`.
- **Сопровождение изменения в столбце.** После проведения таких операций рефакторинга, как "Применение стандартных кодовых обозначений" (с. 188) и "Применение стандартного типа" (с. 192), часто требуется обновить значения, хранящиеся в соответствующем столбце. В частности, вслед за проведением первой из указанных операций рефакторинга часто требуется унифицировать значения, используемые в столбце, например, заменить значения `"US"`, `"USA"` и `"U.S."` единственным значением, `"USA"`. А при проведении второй из указанных операций рефакторинга часто требуется преобразование значений данных, например из числовых в символьные.
- **Исправление транзакционных данных.** После обнаружения дефектов в развернутом приложении или в базе данных часто возникают недопустимые результаты, которые требуют корректировки в составе работ по устранению обнаруженного дефекта (дефектов). Например, может оказаться, что приложением внесены в таблицу `Charges` неправильные значения процентов, относящиеся к какому-то клиенту, исходя из данных в таблице `InterestRate`; а в ходе устранения обнаруженного нарушения в работе необходимо обновить таблицы `InterestRate` и `Charges`, для того чтобы в них отражались правильные значения.

Потенциальные преимущества и недостатки

Задача обновления данных в таблицах может оказаться сложной, особенно если приходится обновлять справочные данные. Например, предположим, что таблица `Account` ссылается на данные в таблице `AccountType`. Данные, внесенные в результате обновления в таблицу `AccountType`, должны содержать значения, которые имеют смысл применительно к данным, содержащимся в таблице `Account`. При обновлении данных необходимо обновлять только строки с правильными данными, причем все эти строки.

Процедура обновления схемы

В результате проведения этого преобразования схема базы данных не изменяется.

Процедура переноса данных

Если количество строк, требующих обновления, невелико, то, по-видимому, достаточно применить простой сценарий SQL, который обновляет целевую таблицу (таблицы). Если же объем данных достаточно велик, то возникает необходимость в использовании более сложного подхода, например, основанного на применении инструментальных средств типа ETL (“Extract–Transform–Load — извлечь–преобразовать–загрузить”), особенно если для получения данных на основе существующих данных таблицы применяются сложные алгоритмы. Ниже перечислены важные вопросы, которые требуют рассмотрения в связи с реализацией рассматриваемого преобразования.

- Должно ли быть предусмотрено получение исходных данных из существующих прикладных таблиц, или данные предоставляются пользователями на деловом предприятии?
- Утверждены ли вводимые в базу данных значения лицом (лицами), заинтересованным в разработке проекта?
- Какие строки должны быть обновлены?
- Какие столбцы с данными этих строк должны быть обновлены?
- Каковы существующие зависимости?

На рис. 11.5 показано, как внести обновления в таблицу `AccountType` с учетом необходимости представления брокерских счетов. Рассматриваемая таблица `AccountType` поддерживает новое соглашение об именовании, которое вначале должно быть проверено, а затем развернуто на производстве. В следующем коде приведены операторы DML, предназначенные для обновления данных в таблице `AccountType`:

```
UPDATE AccountType SET Name = 'Chequing'
  WHERE AccountTypeID=1;

UPDATE AccountType SET Name = 'Private Banking'
  WHERE AccountTypeID=3;
```

Процедура обновления программ доступа

Если операция рефакторинга “Обновление данных” применяется вследствие осуществления какой-то другой операции рефакторинга базы данных, то должны быть также обновлены внешние программы с учетом этой операции рефакторинга. Если же операция рефакторинга “Обновление данных” применяется в связи с изменением значений данных в существующей схеме, то, возможно, потребуется проведение корректировок во внешнем коде. Прежде всего может потребоваться откорректировать операторы `SELECT`, для того чтобы в конструкциях `WHERE` этих операторов были заданы правильные значения. Например, допустим, что в связи с изменением деловой терминологии необходимо откорректировать значение `'Private'` в столбце `AccountType.Name`, заменив его значением `'Private Banking'`.

Аналогичным образом, может возникнуть необходимость откорректировать исходный код, предназначенный для проверки допустимости значений атрибутов данных. Например, предположим, что в коде приложения предпринимается попытка проверить, содержится ли в столбце `AccountType` значение `'Private'`. А теперь это кодовое обозначение должно быть изменено, и в связи с этим следует предусмотреть в программном обеспечении проверку того, содержится ли в столбце `AccountType` значение `'Private Banking'`.

В следующем определении представления показано, как должны быть откорректированы затронутые этим изменением фрагменты кода приложения после изменения типа счета с переходом от `'Private'` к `'Private Banking'`:

```
// Представление до рефакторинга
CREATE OR REPLACE VIEW PrivateAccounts AS
SELECT
  Account.AccountId,
  Account.CustomerId,
  Account.StartDate,
  Account.Balance,
  Account.isPrimary
FROM
  Account, AccountType
WHERE
  Account.AccountTypeId = AccountType.AccountTypeId
  AND AccountType.Name = 'Private'
;

// Представление после рефакторинга
CREATE OR REPLACE VIEW PrivateAccounts AS
SELECT
  Account.AccountId,
  Account.CustomerId,
  Account.StartDate,
  Account.Balance,
  Account.isPrimary
FROM
  Account, AccountType
WHERE
  Account.AccountTypeId = AccountType.AccountTypeId
  AND AccountType.Name = 'Private Banking'
;
```

Приложение А

Обозначения языка моделирования данных UML

В этом приложении кратко рассматривается система обозначений физического моделирования данных, используемая во всей этой книге. Авторы используют подмножество системы обозначений, описанное в профиле универсального языка моделирования (Unified Modeling Language — UML), первоначально представленном в книге *Agile Database Techniques* [**4**], а теперь сопровождаемом в электронном виде на узле `www.agiledata.org/essays/umlDataModelingProfile.html`.

Основная система обозначений для таблиц, применяемых в схеме базы данных, показана на рис. А.1. Таблицы отображаются в виде прямоугольников с одним, двумя или тремя разделами. Первый раздел, содержащий имя таблицы, является обязательным. Два других раздела необязательны; во втором разделе содержится перечень столбцов таблицы, а в третьем — перечень триггеров, связанных с таблицей (если таковые имеются). В списке столбцов обязательными являются только имена; во всей данной книге ради упрощения чаще всего приводятся имена, но не указываются типы столбцов. Если столбец является частью ключа, за ним следует один или несколько стереотипов UML, описанных в табл. А.1.

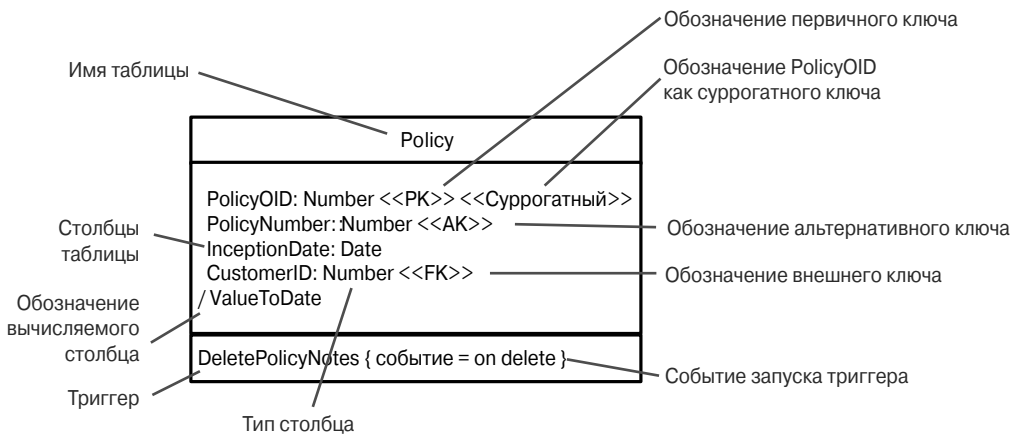


Рис. А.1. Система обозначений, применяемая для моделирования таблиц

Таблица А.1. Стереотипы для ключей

Стереотип	Назначение
PK	Указывает, что столбец является частью первичного ключа для таблицы
FK	Указывает, что столбец является частью внешнего ключа к другой таблице
AK	Указывает, что столбец является частью альтернативного ключа, иногда называемого вторичным ключом
Естественный	Указывает, что ключ представляет собой естественное свойство сущности (например, `Policy`), хранящееся в таблице. Этот стереотип назначается редко; если ключевой столбец не обозначен как суррогатный, предполагается, что он является естественным
Суррогатный	Указывает, что ключ — искусственный (не естественный)

Предусмотрена возможность указать дополнительную информацию, относящуюся к ключам (рис. А.2). В таблице `PolicyNotes` имеются три ключа — первичный ключ, состоящий из столбцов `PolicyNumber` и `NoteNumber`, первый альтернативный ключ, состоящий из столбцов `PolicyOID` и `NoteNumber`, и второй альтернативный ключ в виде столбца `PolicyNoteOID`. Если ключ является составным (иными словами, состоит из нескольких столбцов), то может оказаться важным требование указывать порядок столбцов, относящихся к ключу, чтобы можно было определить должным образом соответствующие индексы. Порядок обозначается с помощью значения `order` (или позиция). Например, можно отметить, что столбец `PolicyNumber` является первым столбцом в составе первичного ключа, а `NoteNumber` — вторым столбцом. Безусловно, из-за применения обозначений порядка столбцов схема становится более громоздкой, поэтому порядок столбцов следует указывать только в случае необходимости.

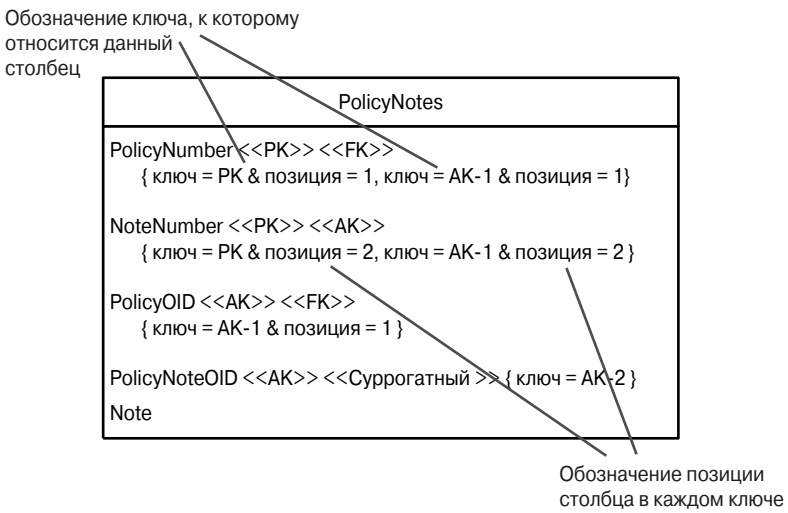


Рис. А.2. Система обозначений, применяемая при моделировании с указанием дополнительных сведений о ключах

Связи, часто называемые *ассоциациями*, моделируются с помощью сплошных линий, проведенных между двумя таблицами. На рис. А.3 показано, что клиент может иметь от нуля или больше полисов и что каждый полис принадлежит единственному клиенту. Острие стрелки рядом с меткой `owns` на связи между таблицами `Customer` и `Account` указывает направление, в котором следует читать эту связь; это — необязательный символ, который должен использоваться только в том случае, если неясно, в каком порядке должны располагаться слова в текстовом прочтении "сущность–связь–сущность". Общепринятое соглашение состоит в том, что метка должна записываться так, чтобы она имела смысл при чтении слева направо или сверху вниз, в зависимости от ориентации линии [7]. Схема, приведенная на рисунке, позволяет определить, что клиенты могут иметь от нуля или больше полисов (поскольку на линии связи рядом с таблицей `Policy` стоит указатель кратности `0..*`) и что любой конкретный полис может принадлежать только одному клиенту (как показывает еще один указатель кратности). Итоговые данные о различных указателях кратности приведены в табл. А.2.

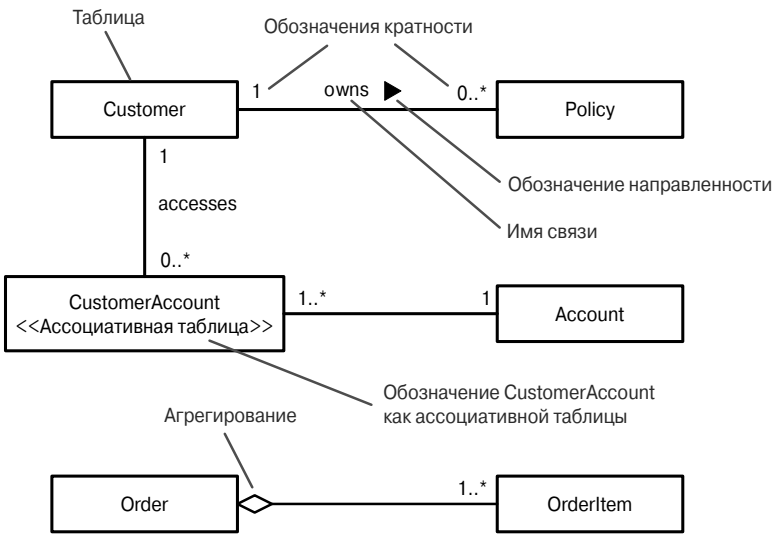


Рис. А.3. Система обозначений, применяемая при моделировании связей

Таблица А.2. Обозначение кратности в языке UML

Кратность	Обозначение
0..1	Нуль или один
1	Только один
0..*	Нуль или больше
1..*	Один или больше
*	Один или больше
n	Только n (где n > 1)
0..n	От нуля до n (где n > 1)
1..n	От одного до n (где n > 1)

На основании данных, приведенных на рис. А.3, можно также сделать вывод, что клиент имеет доступ к счетам, представленным в количестве от нуля или больше, и что к каждому счету имеют доступ от одного или больше клиентов. Безусловно, между сущностями клиента и счета имеется ассоциация "многие ко многим", но ее нельзя непосредственно реализовать в реляционной базе данных с помощью собственных средств базы данных, поэтому для поддержки этой связи добавлена ассоциативная таблица `CustomerAccount`. Кроме того, позиция заказа представляет собой часть одного заказа, а заказ состоит из одной или нескольких позиций заказа. Ромб на конце линии указывает, что связь между таблицами `Order` и `OrderItem` — это одна из связей в составе способа агрегирования, называемая также ассоциацией "part of" (часть). Если рядом с ромбом не приведено обозначение кратности, подразумевается, что кратность равна 1. На рис. А.4 приведено еще несколько примеров связей между таблицами и показано, как читать эти связи.

Рис. А.4. Примеры связей

На рис. А.5 кратко показана система обозначений для некоторых других широко применяемых понятий базы данных, указанных ниже.

- **Хранимые процедуры.** Хранимые процедуры должны быть представлены в прямоугольной области с двумя разделами. В верхнем разделе указаны имя базы данных и стереотип хранимых процедур. В нижнем разделе приведен перечень сигнатур хранимых процедур (или функций) в базе данных.
- **Индексы.** При моделировании индексы обозначаются с помощью прямоугольника с указанием имени индекса, стереотипа `Index` (или Индекс) и отношений зависимости, указывающих на столбец (столбцы), на которых основан индекс.
- **Представления.** Представление изображается в виде прямоугольной области с двумя разделами. В верхнем разделе должны быть указаны имя представления и стереотип `View` (или Представление) . В нижнем разделе, который является необязательным, перечислены столбцы, содержащиеся в представлении.

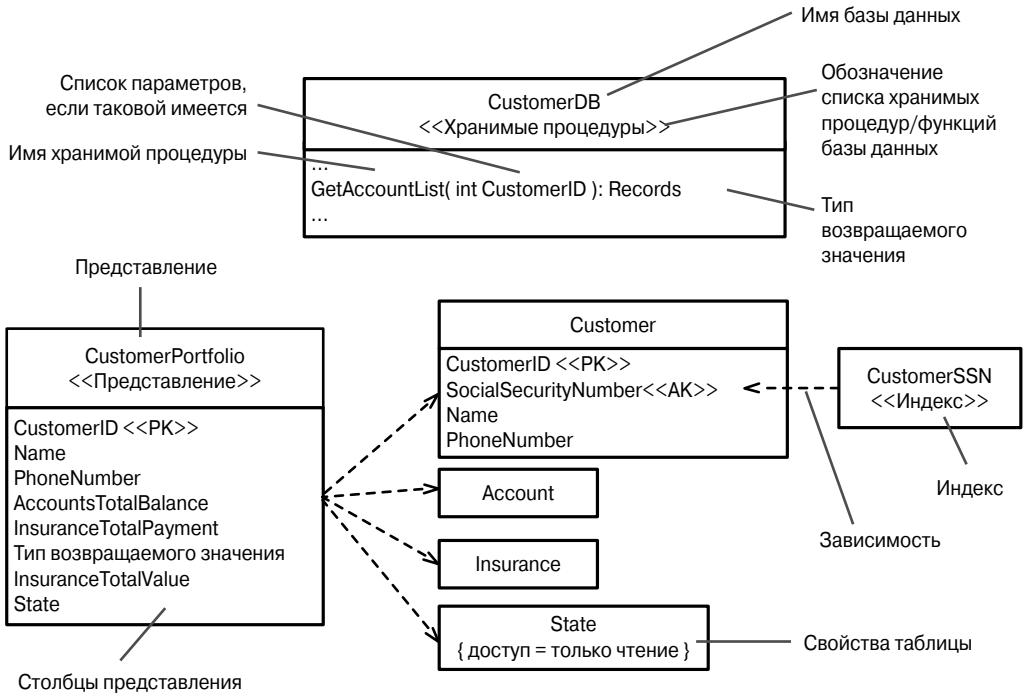


Рис. А.5. Система обозначений, применяемая для моделирования хранимых процедур, представлений и индексов

Приложение Б

Глоссарий

В этом глоссарии даны определения основных терминов, используемых в данной книге.

- **XUnit.** Семейство инструментальных средств блочного тестирования, которое включает инструменты JUnit для Java, VBUnit для Visual Basic, NUnit для .NET и OUnit для Oracle.
- **Адаптивная разработка на основе модели (Agile Model-Driven Development — AMDD).** Подход к разработке, предусматривающий проведение большого количества итераций, в котором вначале создаются адаптивные модели и только после этого начинается написание исходного кода.
- **Адаптивная разработка программного обеспечения.** Эволюционный подход к разработке, позволяющий достичь высокой степени взаимодействия участников, направленный на своевременное создание высококачественного, проверенного программного обеспечения, которое соответствует самым строгим требованиям лиц, заинтересованных в этой работе.
- **Адаптивное моделирование (Agile Modeling — AM).** Хаордическая (chaordic — вносящая порядок в хаос) методология, основанная на сложившейся практике, которая показывает, как добиться высокой эффективности при моделировании и разработке документации.
- **Адаптивный метод разработки системы (Dynamic System Development Method — DSDM).** Адаптивный метод, представляющий собой формализацию методов оперативной разработки приложений (Rapid Application Development — RAD), применявшихся в 1980-х годах.
- **Адаптивный унифицированный процесс (Agile Unified Process — AUP).** Конкретизация унифицированного процесса (Unified Process — UP), в которой применяются такие распространенные адаптивные подходы, как рефакторинг базы данных, разработка на основе тестирования и адаптивная разработка на основе модели.
- **Артефакт.** Документ, модель, файл, диаграмма или другой элемент, который производится, модифицируется или используется в ходе разработки, эксплуатации или сопровождения системы.
- **База данных с несколькими приложениями.** База данных, доступ к которой получает несколько приложений, причем одно или несколько из этих приложений находится за пределами контроля специалистов, занимающихся эксплуатацией базы данных.

- **База данных с одним приложением.** База данных, доступ к которой осуществляется с помощью единственного приложения, находящегося под контролем той же группы, которая контролирует базу данных.
- **Демонстрационная специализированная среда.** Техническая среда, предназначенная для развертывания программного обеспечения в целях его демонстрации лицам, не относящимся непосредственно к группе разработчиков.
- **Извлечение, преобразование и загрузка (Extract-Transform-Load — ETL).** Один из подходов к решению задачи перемещения данных из одного источника в другой, в ходе которого осуществляется "очистка" данных.
- **Инкрементная разработка программного обеспечения.** Подход к разработке программного обеспечения, позволяющий организовать создание проекта в виде нескольких выпусков вместо одного "всеобъемлющего" выпуска.
- **Информационная семантика.** Смысл информации в базе данных с точки зрения пользователей этой информации.
- **Итеративная разработка программного обеспечения.** Подход к разработке, не предусматривающий последовательную организацию, который позволяет заниматься в какое-то время определением требований, в другое время — моделированием, а также выбирать наиболее подходящее время для программирования или тестирования.
- **Итерация.** Период времени, часто составляющий одну-две недели, в течение которого ведется подготовка исходного кода рабочего программного обеспечения. Именуется также *циклом разработки*.
- **Каскадная разработка.** *См.* Последовательная разработка.
- **Концептуальная модель/модель доменов.** Модель, на которой отображаются основные деловые сущности и связи между ними, а также, в случае необходимости, атрибуты или назначения этих сущностей.
- **Метод Scrum.** Адаптивный метод, в основном сосредоточенный на управлении проектом и требованиями. Метод Scrum часто применяется совместно с подходом XP (экстремальное программирование).
- **Метод адаптивных данных (Agile Data — AD).** Совокупность принципов и подходов, позволяющих определить способы осуществления разработок, ориентированных на данные, в адаптивной форме.
- **Метод.** Применительно к базе данных — хранимая процедура, хранимая функция или триггер.
- **Моделирование на основе мозгового штурма.** Краткий, но напряженный этап моделирования, часто имеющий продолжительность от 5 до 15 минут, в ходе которого два или несколько человек совместно работают над исследованием части задачи или проблемной области решения. Сеансы моделирования на основе мозгового штурма должны непосредственно следовать за периодами разработки кода (которые часто занимают по своей продолжительности несколько часов или суток).
- **Недостаток базы данных.** Общая категория проблем, связанных со схемой базы данных, которая указывает на необходимость проведения рефакторинга базы данных.

- **Недостаток кода.** Общая категория проблем, связанных с исходным кодом, которая указывает на необходимость проведения рефакторинга кода.
- **Объект доступа к данным (DAO).** Класс, в котором реализован необходимый код доступа к базе данных, позволяющий обеспечить работу с соответствующим классом деловой сущности, хранящимся в базе данных.
- **Объектно-реляционное отображение (Object-relational mapping — ORM).** Определение связи (связей) между компонентами объектной схемы, представляющими данные (например, классами Java), и схемой реляционной базы данных.
- **Оперативная разработка приложений (Rapid Application Development — RAD).** Подход к разработке программного обеспечения, который по своему характеру является в значительной степени эволюционным и, как правило, предусматривает выполнение значительной части работы в форме подготовки прототипов пользовательского интерфейса.
- **Операция рефакторинга базы данных.** Простое изменение в схеме базы данных, которое способствует улучшению проекта базы данных, сохраняя при этом функциональную и информационную семантику; иными словами, проведение операции рефакторинга не должно повлечь за собой добавление новых функциональных средств, нарушение в работе существующих функциональных средств, добавление новых данных или изменение смысла существующих данных.
- **Операция рефакторинга.** Простое изменение в исходном коде, позволяющее сохранить функциональную семантику кода. Проведение операции рефакторинга не должно приводить ни к расширению, ни к сужению функциональных возможностей кода. Рефакторинг просто способствует улучшению проекта кода.
- **Отчетное совещание.** Краткое совещание, в ходе которого отдельным членам группы предоставляется слово для выступления, чтобы они могли отчитаться о состоянии выполнения задач, порученных им накануне, сообщить о своих планах на сегодняшний день, рассказать о возникших проблемах, охарактеризовать внесенные ими изменения в архитектуру, а также поделиться информацией о других важных событиях, о которых они должны поставить в известность свою группу.
- **Переходный период.** Время, в течение которого параллельно поддерживается и старая, и новая схема. Применяется также название период эксплуатации устаревшего программного обеспечения.
- **Период эксплуатации устаревшего программного обеспечения.** См. *Переходный период*.
- **Подходящий интервал развертывания.** Конкретный период времени, в течение которого допускается развертывание системы на производстве. Часто именуется *подходящим интервалом выпуска*.
- **Последовательная разработка.** Подход, предусматривающий создание весьма детализированных моделей до того, как будет “разрешена” реализация этих моделей. Известный также под названием каскадной разработки.
- **Преобразование базы данных.** Изменение в схеме базы данных, которое может приводить или не приводить к изменению семантики схемы. Любая операция рефакторинга базы данных представляет собой разновидность операции преобразования базы данных.

- **Производственная среда.** Техническая среда, в которой конечные пользователи эксплуатируют одну или несколько систем.
- **Разработка на основе первоочередного тестирования (Test-First Development — TFD).** Эволюционный подход к разработке, в котором необходимо вначале написать тест, выполнение которого оканчивается неудачей, и только после этого приступать к созданию такого нового функционального кода, в результате применения которого выполнение теста завершится успешно. Этот подход известен также как программирование на основе первоочередного тестирования (Test-First Programming).
- **Разработка на основе тестирования (Test-Driven Development — TDD).** Сочетание методов TFD и рефакторинга.
- **Разработка на основе функций (Feature-Driven Development — FDD).** Адаптивный метод разработки, предусматривающий проведение коротких итераций, которые определяются с учетом общей объектной модели доменов и функций (в целях реализации требований к небольшим изменениям).
- **Регрессионное тестирование базы данных.** Рефакторинг проводится таким образом, чтобы можно было действительно гарантировать нормальное функционирование схемы базы данных; для этого разрабатывается тестовый набор, который в дальнейшем регулярно выполняется применительно к базе данных.
- **Регрессионный тестовый набор.** Набор тестов, регулярно выполняемый по отношению к системе для проверки того, что система работает в соответствии с эталонными результатами тестов.
- **Рефакторинг архитектуры.** Изменение, способствующее в целом улучшению взаимодействия внешних программ с базой данных.
- **Рефакторинг базы данных.** Процесс, позволяющий развивать существующую схему базы данных, проводя каждый раз небольшие изменения, осуществляемые в целях повышения качества проекта базы данных без изменения ее семантики.
- **Рефакторинг качества данных.** Изменение, которое позволяет повысить качество информации, хранящейся в базе данных.
- **Рефакторинг метода.** Изменение, вносимое в метод, которое способствует повышению его качества. К методам базы данных применимы многие операции рефакторинга кода.
- **Рефакторинг ссылочной целостности.** Изменение, которое гарантирует, что указанная в ссылке строка существует в другой таблице, и (или) позволяет обеспечить удаление должным образом строк, которые становятся больше не нужными.
- **Рефакторинг структуры.** Изменения в определении одной или нескольких таблиц или представлений.
- **Рефакторинг.** Методика программирования, позволяющая медленно развивать код во времени, применяя эволюционный подход к программированию.
- **Связность.** Мера зависимости между двумя объектами; чем более тесно связаны два предмета, тем выше вероятность того, что изменение в одном из них потребует внесения изменений в другой.

- **Семейство методологий Crystal.** Легко приспосабливаемое к конкретным обстоятельствам семейство адаптивных методологий, предусматривающих участие человека, которые были разработаны Алистером Кокберном (Alistair Cockburn).
- **Специализированная среда интеграции проектов.** Техническая среда, в которой осуществляется компиляция и тестирование кода, создаваемого членами всех групп проектировщиков.
- **Специализированная среда разработки.** Техническая среда, в которой специалисты в области информационной технологии создают, тестируют и выполняют сборку программного обеспечения.
- **Специализированная среда.** Полностью работоспособная среда, в которой могут осуществляться сборка, тестирование и (или) эксплуатация определенной системы.
- **Ссылочная целостность.** Правила, обеспечивающие поддержку действительных ссылок, направленных от одной сущности к другой. Если сущность A ссылается на сущность B, то сущность B должна существовать. В случае удаления сущности B должны быть также удалены все ссылки на сущность B.
- **Стереотип.** Конструкция UML, которая характеризует общепринятое назначение определенного элемента моделирования. Стереотипы используются для расширения состава средств языка UML регламентированным способом.
- **Схема базы данных.** Структурные компоненты базы данных, такие как таблицы и определения представлений, а также функциональные компоненты, такие как методы базы данных.
- **Транзакция.** Единица работы, которая либо полностью выполняется успешно, либо полностью оканчивается неудачей. Транзакция может состоять из одной или нескольких операций обновления объекта, одной или нескольких операций чтения, одной или нескольких операций удаления либо включать произвольные комбинации этих операций.
- **Триггер.** Метод базы данных, автоматически вызываемый в результате выполнения операций языка манипулирования данными (Data Manipulation Language — DML) в рамках механизма обеспечения перманентности.
- **Универсальный язык моделирования (Unified Modeling Language — UML).** Определение стандартного языка моделирования для объектно-ориентированного программного обеспечения, включающее определение системы обозначений для моделирования и семантики применения этих обозначений, которое было создано корпорацией Object Management Group (OMG).
- **Унифицированный процесс компании Rational (Rational Unified Process — RUP).** Строго формализованный, четырехэтапный процесс разработки программного обеспечения, созданный компанией IBM Rational, который по своему характеру является эволюционным.
- **Функциональная семантика.** Общий смысл реализации функциональных средств в базе данных.
- **Эволюционная разработка программного обеспечения.** Подход, обеспечивающий итеративную и инкрементную организацию работы.

- **Эволюционное моделирование данных.** Процесс, позволяющий моделировать компоненты системы, связанные с обработкой данных, итеративно и инкрементно, что обеспечивает развитие схемы базы данных в таком же темпе, как и прикладного кода.
- **Экстремальное программирование (Extreme Programming — XP).** Формализованный и регламентированный метод адаптивной разработки, позволяющий сосредоточиться на осуществлении важных действий, необходимых для создания программного обеспечения.
- **Этап применения на производстве.** Часть жизненного цикла системы, в которой осуществляется эксплуатация системы ее конечными пользователями.
- **Язык манипулирования данными (DML).** Команды, поддерживаемые базой данных, которая обеспечивают доступ к данным, содержащимся в базе данных, в частности, обеспечивают создание, выборку, обновление и удаление этих данных.

Приложение В

Литература

1. Agile Alliance (2001a) *Manifesto for Agile Software Development*. www.agilealliance.org.

2. Agile Alliance (2001b) *Principles: The Agile Alliance*. www.agilealliance.org/principles.html.

3. Ambler S. W. (2002) *Agile Modeling: Best Practices for the Unified Process and Extreme Programming*. New York: John Wiley & Sons. www.ambysoft.com/agileModeling.html.

4. Ambler S. W. (2003) *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York: John Wiley & Sons. www.ambysoft.com/agileDatabaseTechniques.html.

5. Ambler S. W. (2004) *The Object Primer, 3rd Edition: Agile Model Driven Development with UML 2*. New York: Cambridge University Press. www.ambysoft.com/theObjectPrimer.html.

6. Ambler, S. W. (2005a) *A UML Profile for Data Modeling*. http://www.agiledata.org/essays/umlDataModelingProfile.html.

7. Ambler S. W. (2005b) *The Elements of UML 2.0 Style*. New York: Cambridge University Press. www.ambysoft.com/elementsUMLStyle.html.

8. Ambler S.W. (2005c) *The Agile Unified Process*. www.ambysoft.com/unifiedprocess/agileUP.html.

9. Astels D. (2003) *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall.

10. Beck K. (2003) *Test Driven Development: By Example*. Boston, MA: Addison-Wesley.

11. Boehm B. W. and Turner R. (2003) *Balancing Agility with Discipline: A Guide for the Perplexed*. Reading MA: Addison-Wesley Longman, Inc.

12. Burry C. and Mancusi D. (2004) *How to Plan for Data Migration*. Computerworld, www.computerworld.com/databasetopics/data/story/0,10801,93284,00.html.

13. Celko J. (1999) *Joe Celko's Data & Databases: Concepts in Practice*. San Francisco: Morgan Kaufmann.

14. Cockburn A. (2002) *Agile Software Development*. Reading, MA: Addison-Wesley Longman, Inc.

15. Date C. J. (2001) *An Introduction to Database System, 7/e*. Reading, MA: Addison-Wesley Longman, Inc. (К. Дж. Дейт. *Введение в системы баз данных*. Пер. с англ., М: ИД “Вильямс”, 2006 г.)

16. Feathers M. (2004) *Working Effectively with Legacy Code*. Boston: Addison Wesley.

17. Fowler M. (1999) *Refactoring: Improving the Design of Existing Code*. Menlo Park, CA: Addison-Wesley Longman.

18. Fowler M. and Sadalage P. (2003) *Evolutionary Database Design*. `http://martinfowler.com/articles/evodb.html`.

19. Halpin T. A. (2001) *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. San Francisco: Morgan Kaufmann.

20. Hay D. C. (1996) *Data Model Patterns: Conventions of Thought. New York*: Dorset House Publishing.

21. Hay D. C. (2003) *Requirements Analysis: From Business Views to Architecture*. Upper Saddle River, NJ: Prentice Hall, Inc.

22. Hernandez M. J. and Viescas J. L. (2000) *SQL Queries for Mere Mortals: A Hands-on Guide to Data Manipulation in SQL*. Reading, MA: Addison-Wesley.

23. Kerievsky J. (2004) *Refactoring to Patterns*. Boston: Addison-Wesley. (Джошуа Кериевски. *Рефакторинг с использованием шаблонов*. Пер. с англ., М: ИД "Вильямс", 2006 г.)

24. Kyte T. (2005) *Avoiding Mutating Tables*. `http://asktom.oracle.com/~tkyte/Mutate/`.

25. Larman C. (2004) *Agile and Iterative Development: A Manager's Guide*. Boston: Addison-Wesley.

26. Loshin D. (2004a) *Issues and Opportunities in Data Quality Management Coordination*. DM Review Magazine, April 2004.

27. Loshin D. (2004b) *More on Information Quality ROI. The Data Administration Newsletter (TDAN.com)*. July 2004. `www.tdan.com/nwt_issue29.htm`.

28. Manns M. L. and Rising L. (2005) *Fearless Change: Patterns for Introducing New Ideas*. Boston: Pearson Education, Ltd.

29. Muller R. J. (1999) *Database Design for Smarties: Using UML for Data Modeling*. San Francisco: Morgan Kaufmann.

30. Mullins C. S. (2002) *Database Administration: The Complete Guide to Practices and Procedures*. Reading, MA: Addison-Wesley Longman, Inc.

31. Pascal F. (2000) *Practical Issues in Database Management: A Reference for the Thinking Practitioner*. Upper Saddle River, NJ: Addison-Wesley Professional.

32. Sadalage P. and Schuh P. (2002) *The Agile Database: Tutorial Notes*. Paper presented at XP/Agile Universe 2002. Retrieved November 12, 2003 from `http://www.xpuniverse.com`.

33. Seiner R. (2004) *Data Stewardship Performance Measures*. The Data Administration Newsletter, July 2004. `www.tdan.com/i029fe01.htm`.

34. Williams L. and Kessler R. (2002) *Pair Programming Illuminated*. Boston, MA: Addison-Wesley.

Приложение Г

Список операций рефакторинга и операций преобразования

Список операций рефакторинга и операций преобразования приведен ниже.

- **Введение вычислительного метода** (с. 268). Введение нового метода, как правило, хранимой функции, реализующей вычисления, в которых используются данные, хранимые в базе данных.
- **Введение вычисляемого столбца** (с. 120). Введение нового столбца, значения в котором формируются на основе вычислений, с использованием данных из одной или нескольких таблиц.
- **Введение заданного по умолчанию значения** (с. 213). Предоставление базе данных возможности подставлять заданное по умолчанию значение в существующий столбец таблицы.
- **Введение индекса** (с. 271). Введение нового индекса уникального или неуникального типа.
- **Введение каскадного удаления** (с. 239). Обеспечение того, чтобы база данных автоматически удаляла соответствующие "дочерние" записи после удаления "родительской" записи.
- **Введение нового столбца** (с. 321). Введение нового столбца в существующей таблице.
- **Введение новой таблицы** (с. 323). Введение новой таблицы в существующую базу данных.
- **Введение общего формата** (с. 210). Применение согласованного формата ко всем значениям данных в существующем столбце таблицы.
- **Введение ограничения столбца** (с. 207). Введение ограничения столбца на существующей таблице.
- **Введение переменной** (с. 310). Присваивание результатов вычисления выражения или частей выражения временной переменной, имя которой поясняет ее назначение.
- **Введение представления** (с. 325). Создание представления на основе существующих таблиц базы данных.

- **Введение программного удаления** (с. 246). Введение в существующую таблицу флажка, которая указывает, что строка была удалена, вместо фактического удаления строки.
- **Введение суррогатного ключа** (с. 123). Замена существующего естественного ключа суррогатным ключом.
- **Введение таблицы только для чтения** (с. 273). Создание допускающего только чтение хранилища данных на основе существующих таблиц в базе данных.
- **Введение триггера для накопления исторических данных** (с. 250). Введение нового триггера для перехвата информации об изменениях в данных в целях накопления хронологических сведений или проведения аудита.
- **Введение физического удаления** (с. 243). Удаление существующего столбца, который указывает, что строка была удалена, и переход вместо этого к фактическому удалению строк.
- **Вставка данных** (с. 317). Вставка данных в существующую таблицу.
- **Декомпозиция условного выражения** (с. 306). Извлечение методов из условий.
- **Добавление зеркальной таблицы** (с. 259). Создание зеркальной таблицы (точного дубликата существующей таблицы из одной базы данных) в другой базе данных.
- **Добавление метода чтения** (с. 263). Введение метода (в этом случае хранимой процедуры), позволяющего реализовать операцию выборки данных из базы данных, представляющих от нуля или больше деловых сущностей.
- **Добавление методов CRUD** (с. 255). Введение хранимых процедур (методов) для реализации операций создания, выборки, обновления и удаления (Creation, Retrieval, Update, Deletion — CRUD) данных, представляющих деловую сущность.
- **Добавление ограничения внешнего ключа** (с. 229). Добавление ограничений внешнего ключа в существующую таблицу для поддержки связи с другой таблицей.
- **Добавление параметра** (с. 300). Добавление параметра в связи с тем, что для существующего метода требуется информация, которая до сих пор в него не передавалась.
- **Добавление поисковой таблицы** (с. 185). Создание поисковой таблицы для существующего столбца.
- **Добавление триггера для вычисляемого столбца** (с. 234). Введение нового триггера, предназначенного для обновления значений, содержащихся в вычисляемом столбце.
- **Замена вложенного условного выражения защитными конструкциями** (с. 314). Удаление вложенных операторов проверки условий `if` и введение вместо них ряда отдельных операторов `IF`.
- **Замена данных типа LOB таблицей** (с. 154). Замена столбца крупного объекта (Large OBject — LOB), который содержит структурированные данные, новой таблицей или столбцами той же таблицы.
- **Замена кодового обозначения типа флажками свойств** (с. 222). Замена столбца с кодами отдельными флажками свойств, обычно реализуемыми как булевы столбцы, в пределах одного и того же столбца таблицы.

- **Замена метода (методов) представлением** (с. 287). Создание представления на основе одного или нескольких существующих методов базы данных (хранимых процедур, хранимых функций или триггеров), которые заданы в этой базе данных.
- **Замена параметра явно заданными методами** (с. 304). Создание отдельного метода для каждого значения параметра.
- **Замена подстановки литерала поиском в таблице** (с. 312). Замена констант, заданных в коде, значениями, полученными из таблиц базы данных.
- **Замена представления методом (методами)** (с. 290). Замена существующего представления одним или несколькими существующими методами (хранимыми процедурами, хранимыми функциями или триггерами), хранящимися в той же базе данных.
- **Замена связи "один ко многим" ассоциативной таблицей** (с. 163). Замена связи "один ко многим" между двумя таблицами ассоциативной таблицей.
- **Замена столбца** (с. 160). Замена существующего неключевого столбца новым столбцом.
- **Замена суррогатного ключа естественным ключом** (с. 168). Замена суррогатного ключа существующим естественным ключом.
- **Извлечение метода** (с. 307). Преобразование фрагмента кода в метод, имя которого может служить пояснением к назначению метода.
- **Инкапсуляция таблицы в представление** (с. 266). Инкапсуляция средств доступа к существующей таблице в представление.
- **Использование официально заданного источника данных** (с. 292). Переход к использованию для рассматриваемой сущности официально утвержденного источника данных вместо используемого в настоящее время.
- **Консолидация условного выражения** (с. 305). Объединение последовательности операций проверки условия в единственном условном выражении и извлечение полученного выражения.
- **Обновление данных** (с. 329). Обновление данных, относящихся к существующей таблице.
- **Осуществление стратегии консолидированных ключей** (с. 197). Определение единой стратегии поддержки ключей для любой сущности и последовательное ее применение во всей базе данных.
- **Параметризация метода** (с. 300). Создание одного метода, в котором определенный параметр используется для представления различных значений.
- **Переименование метода** (с. 302). Переименование существующего метода с присваиванием имени, которое объясняет его назначение.
- **Переименование представления** (с. 152). Переименование существующего представления с присваиванием имени, которое объясняет его назначение.
- **Переименование столбца** (с. 145). Переименование существующего столбца таблицы с присваиванием имени, которое объясняет его назначение.

- **Переименование таблицы** (с. 148). Переименование существующей таблицы с присваиванием имени, которое объясняет ее назначение.
- **Перемещение данных** (с. 219). Перемещение данные, содержащихся в таблице, либо полностью, либо в виде подмножества столбцов таблицы, в другую существующую таблицу.
- **Перемещение столбца** (с. 139). Перемещение столбца таблицы со всеми его данными в другую существующую таблицу.
- **Перенос метода в базу данных** (с. 284). Перебазирование существующего фрагмента прикладного программного обеспечения в базу данных.
- **Перенос метода из базы данных** (с. 280). Перебазирование существующего метода базы данных (хранимой процедуры, хранимой функции или триггера) в приложение (приложения), из которого он до сих пор вызывался.
- **Переупорядочение параметров** (с. 302). Изменение порядка расположения параметров метода.
- **Подстановка алгоритма** (с. 315). Замена тела метода новой реализацией алгоритма.
- **Преобразование столбца в недопускающий NULL-значения** (с. 216). Изменение существующего столбца таким образом, чтобы он не принимал больше какие-либо NULL-значения.
- **Применение стандартного типа** (с. 192). Обеспечение того, чтобы тип данных столбца соответствовал типу данных других подобных столбцов в базе данных.
- **Применение стандартных кодовых обозначений** (с. 188). Применение стандартного набора значений кода к единственному столбцу для обеспечения того, чтобы содержащиеся в нем значения соответствовали значениям подобных столбцов, хранящихся в другом месте базы данных.
- **Разбиение временной переменной** (с. 314). Создание отдельной временной переменной вместо каждого оператора присваивания.
- **Разбиение столбца** (с. 172). Разбиение столбца на один или несколько столбцов в пределах одной и той же таблицы.
- **Разбиение таблицы** (с. 177). Разбиение по вертикали (например, по столбцам) существующей таблицы на одну или несколько таблиц.
- **Слияние столбцов** (с. 130). Объединение двух или нескольких столбцов в одной и той же таблице.
- **Слияние таблиц** (с. 133). Объединение двух или нескольких таблиц в одну таблицу.
- **Удаление параметра** (с. 301). Удаление параметра, который больше не используется в теле метода.
- **Удаление посредника** (с. 311). Обеспечение непосредственного вызова метода в вызывающем коде.
- **Удаление представления** (с. 118). Удаление существующего представления.
- **Удаление столбца** (с. 112). Удаление столбца из существующей таблицы.
- **Удаление таблицы** (с. 117). Удаление существующей таблицы из базы данных.

- **Удаление флажка управления** (с. 311). Использование операции удаления или прерывания вместо переменной, действующей как флажок управления.
- **Уничтожение значения, заданного по умолчанию** (с. 203). Удаление заданного по умолчанию значения, которое предоставлялось базой данных, из существующего столбца таблицы.
- **Уничтожение ограничения внешнего ключа** (с. 238). Удаление ограничения внешнего ключа из существующей таблицы, для того чтобы базой данных больше не предписывалась связь с другой таблицей.
- **Уничтожение ограничения столбца** (с. 201). Удаление ограничения столбца из существующей таблицы.
- **Уничтожение столбца, не допускающего NULL-значений** (с. 205). Изменение существующего столбца, не допускающего NULL-значений, для обеспечения того, чтобы он принимал NULL-значения.

Приложение Д

Отзывы

Ниже приведены отзывы специалистов о данной книге.

- “Эта книга — значительное достижение; в ней показано, как упростить модернизацию схем баз данных, обеспечить эффективное перемещение данных и снизить для специалистов в области баз данных трудоемкость выполнения запросов на внесение изменений, беспрестанно поступающих от клиентов и разработчиков. Основой адаптивного подхода является эволюционное проектирование. Своей книгой Эмблер и Садаладже наглядно продемонстрировали перспективность применения адаптивного подхода для усовершенствования баз данных. Приношу им свои поздравления!”

  Джошуа Кериевски (Joshua Kerievsky), основатель компании Industrial Logic, Inc., автор книги *Рефакторинг с использованием шаблонов*.

- “В этой книге не только изложены основные принципы эволюционной разработки базы данных, но и приведены в большом количестве практические, подробные примеры проведения операций рефакторинга базы данных. Эта книга является буквально необходимой для специалистов-практиков в области баз данных, заинтересованных в проведении адаптивных разработок”.

  Дуг Барри (Doug Barry), президент компании Barry & Associates, Inc.; автор книги *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide*.

- “Эмблеру и Садаладже удалось успешно решить задачу, которая другим техническим писателям показалась просто непосильной. Авторы смогли не только изложить теорию, лежащую в основе рефакторинга базы данных, но и поэтапно описать такие процессы осуществления операций рефакторинга, которые позволяют все тщательно продумать и взять под свой контроль все осуществляемые действия. Но больше всего меня поразило то, что эта книга включает больше 200 страниц с примерами кода и глубокими описаниями практических рекомендаций, позволяющих преодолеть конкретные затруднения, которые часто бывают связаны с осуществлением операций рефакторинга базы данных. Это — не просто еще одна ознакомительная книга, направленная на то, чтобы убедить широкую публику в перспективности излагаемых идей; это — учебник и технический справочник, который должны всегда держать под рукой разработчики, администраторы базы данных и другие специалисти в области обработки данных. Выражаю свою благодарность двум отважным авторам, сумевшим достичь цели, к которой другие не пытались даже приблизиться”.

  Кевин Агуанно, старший руководитель проекта, компания IBM Canada Ltd.

- “Любой специалист, которому довелось работать над проектами, не начинающимися с нуля, оценит значимость вклада, который Скотт и Прамод внесли в усовершенствование жизненного цикла разработки программного обеспечения, выпустив книгу *Рефакторинг баз данных: эволюционное проектирование*. Проблемы, с которыми приходится сталкиваться при внесении изменений в существующие базы данных, отличаются особой сложностью. Безусловно, многие из этих проблем решаются в рамках сложившихся подходов, и работа не останавливается благодаря значительным усилиям администраторов баз данных, но в этой книге показано, как применить адаптивный подход на практике для формализации операций рефакторинга и перехода к эволюционной разработке базы данных. Я намереваюсь широко пропагандировать эту книгу среди администраторов баз данных, поскольку изложенные в ней методы позволяют значительно облегчить их повседневную нагрузку”.

  Джон Керн (Jon Kern)

- “Эта книга превосходна. Она идеально подходит для специалистов в области обработки данных, которым приходится применять в своей работе адаптивные методы разработки и объектную технологию. Книга хорошо организована; в ней освещены все вопросы, касающиеся применения операций рефакторинга баз данных и относящегося к ним кода. Я намереваюсь часто использовать ее как превосходный сборник рекомендаций в ходе разработки и усовершенствования баз данных”.

  Дэвид Р. Херцен (David R. Haertzen), редактор, Центр управления данными компании First Place Software, Inc.

- “Эта превосходная книга позволяет перенести методы адаптивной разработки, основанные на применении рефакторинга, в сферу обработки данных. Она представляет собой практическое руководство, в котором описана методология проведения операций рефакторинга баз данных в конкретной организации и приведены подробные сведения о том, как реализовать отдельные операции рефакторинга. Кроме того, в книге *Рефакторинг баз данных: эволюционное проектирование* подчеркивается важность обеспечения взаимодействия разработчиков и администраторов баз данных. Эта книга должна стать настольным справочником для разработчиков, администраторов баз данных и других специалистов в области обработки данных”.

  Пер Кролл (Per Kroll), руководитель разработки, RUP, IBM; руководитель проекта, Eclipse Process Framework

- “Скотт и Прамод внесли в развитие такого направления, как рефакторинг баз данных, значительный вклад, сравнимый с тем, что было сделано Мартином Фаулером для реализации возможностей рефакторинга кода. Они свели воедино органичный набор процедур, которые в целом могут использоваться для повышения качества базы данных. Эта книга — для всех, кто работает с базами данных”.

  Кен Пью (Ken Pugh), автор книги *Prefactoring*

- “Для специалистов в области обработки данных давно настало время освоить адаптивные методики, и в этом могут помочь Эмблер и Садаладже. Эту книгу обязаны прочесть не только разработчики моделей данных и администраторы, но и специалисты в области разработки программного обеспечения. При создании баз данных и программного обеспечения слишком долго применялись разные подхо-

ды, и эта книга поможет преодолеть барьеры, разделяющие специалистов из той и другой области".

Гэри К. Эванс (Gary K. Evans), горячий приверженец адаптивных подходов, компания Evanetics, Inc.

- "Эволюционное проектирование и рефакторинг уже давно привлекают значительный интерес, и с выпуском книги *Рефакторинг баз данных: эволюционное проектирование* заинтересованность в развитии этих подходов становится еще более ярко выраженной. В этой книге авторы делятся с нами своим опытом в области применения методов и стратегий проведения рефакторинга на уровне базы данных. Описанные ими операции рефакторинга позволяют обеспечить безопасное развитие схемы баз данных даже после развертывания базы данных на производстве. Благодаря этой книге любой разработчик получает возможность внести в базу данных такие изменения, необходимость в проведении которых давно назрела".

  Sven Gorts (Свен Гортс)

- "Рефакторинг базы данных — важная новая тема, а эта книга — важный новаторский вклад, который пойдет на пользу всему сообществу разработчиков".

  Флойд Маринеску (Floyd Marinescu), создатель узлов `InfoQ.com` и `TheServerSide.com`, автор книги *EJB Design Patterns*.

Предметный указатель

S

T

U

X

А

Б

В

К

М

Н

О

П

P

C

Т

У

Ф

Ц

Ш

Э

Я

Научно-популярное издание

Скотт В. Эмблер, Прамодкумар Дж. Садаладж

Рефакторинг баз данных
Эволюционное проектирование

Литературный редактор *И.А. Попова*
Верстка *А.В. Плаксюк*
Художественный редактор *С.А. Чернокозинский*
Корректор *Л.А. Гордиенко*

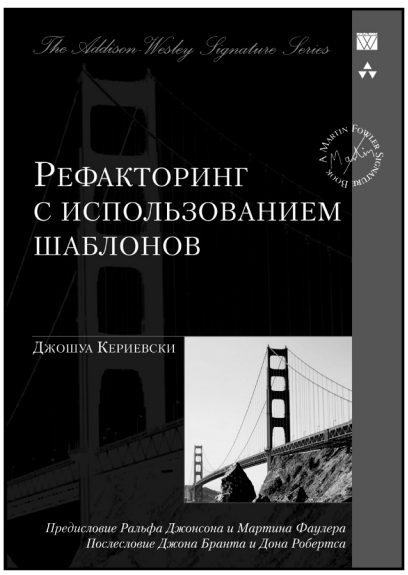
Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 22.03.2007. Формат 70x100/16
Гарнитура Newton. Печать офсетная
Усл. печ. л. 23,0. Уч.-изд. л. 20,3
Тираж 2000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15

РЕФАКТОРИНГ С ИСПОЛЬЗОВАНИЕМ ШАБЛОНОВ

Джошуа Кериевски



www.williamspublishing.com

Данная книга представляет собой результат многолетнего опыта профессионального программиста по применению шаблонов проектирования. Авторский подход к проектированию состоит в том, что следует избегать как недостаточного, так и избыточного проектирования, постоянно анализируя готовый работоспособный код и реорганизуя его только в том случае, когда это приведет к повышению его эффективности, упрощению его понимания и сопровождения. Автор на основании как собственного, так и чужого опыта детально рассматривает различные признаки кода, требующего рефакторинга, описывает, какой именно рефакторинг наилучшим образом подходит для той или иной ситуации, и описывает его механику, подробно разбирая ее на конкретных примерах из реальных задач. Книга может рассматриваться и как учебник по рефакторингу для программиста среднего уровня, и как справочное пособие для профессионала.

ISBN 5-8459-1087-0

в продаже