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.

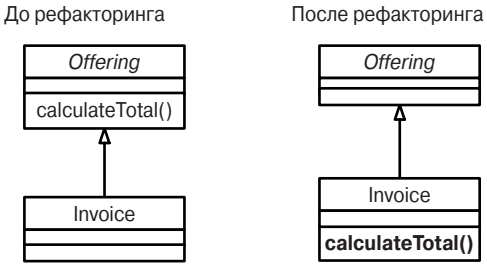

Рис. 2.1. Перенос метода из класса в подкласс

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.

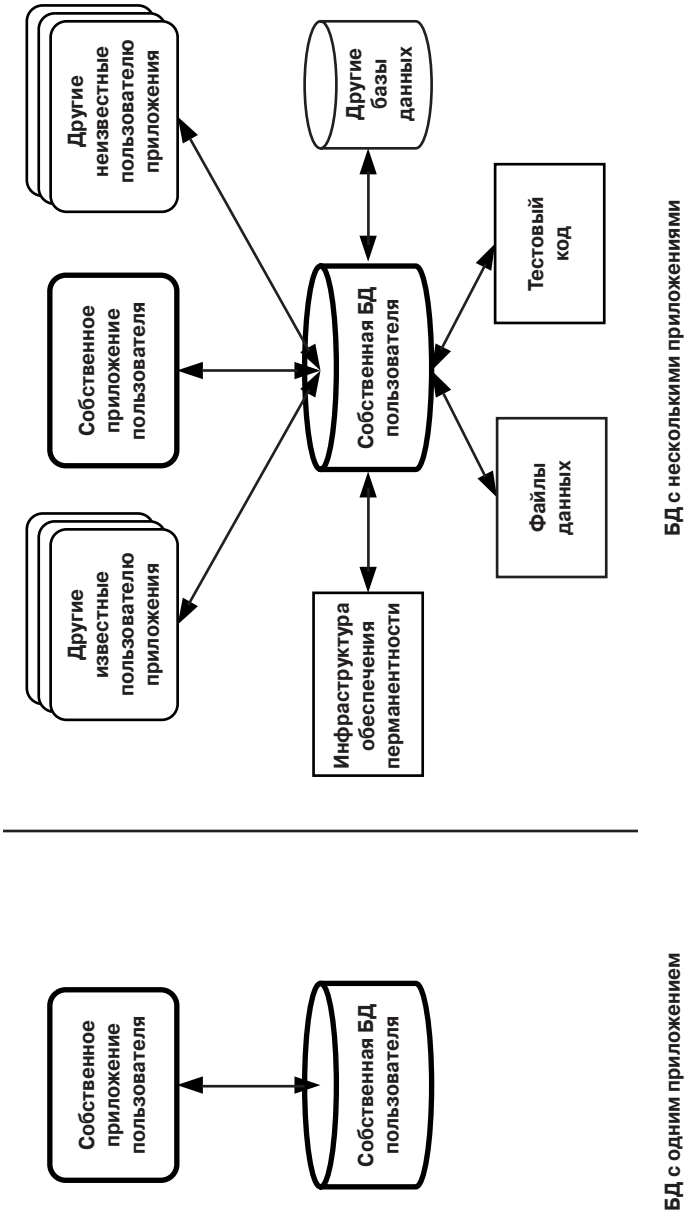

Рис. 2.2. Две категории архитектур базы данных

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).

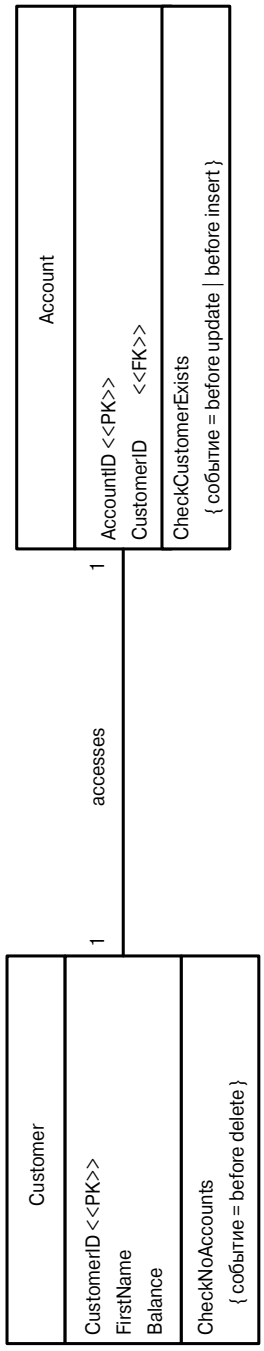

Рис. 2.3. Фрагмент исходной схемы базы данных, относящийся к таблицам `Customer` *и* `Account`

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.

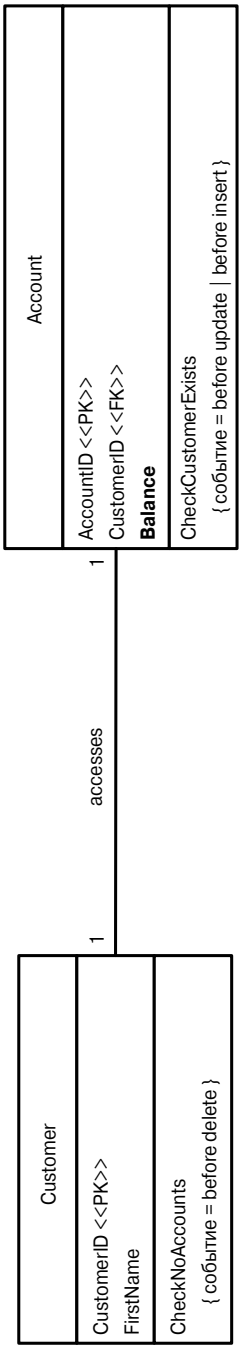

Рис. 2.4. Фрагмент окончательно полученной схемы базы данных, относящийся к таблицам `Customer` *и* `Account`

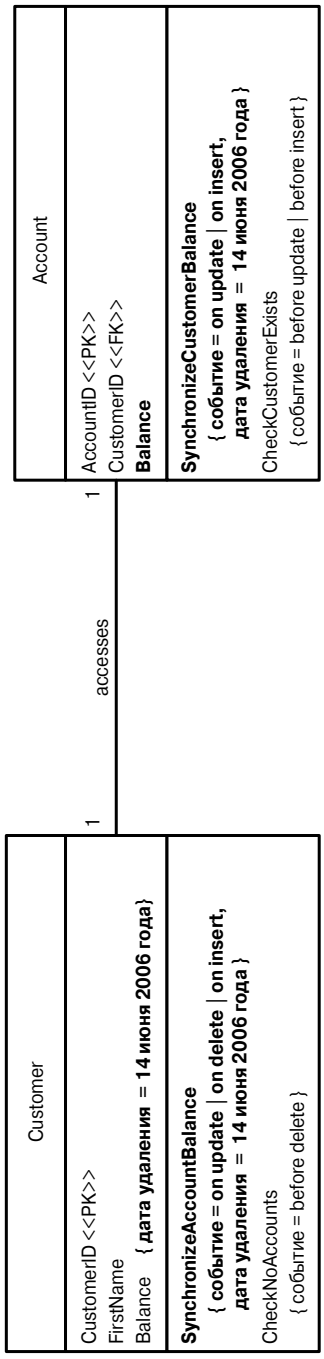

Рис. 2.5. Схема базы данных, применяемая на протяжении переходного периода

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.

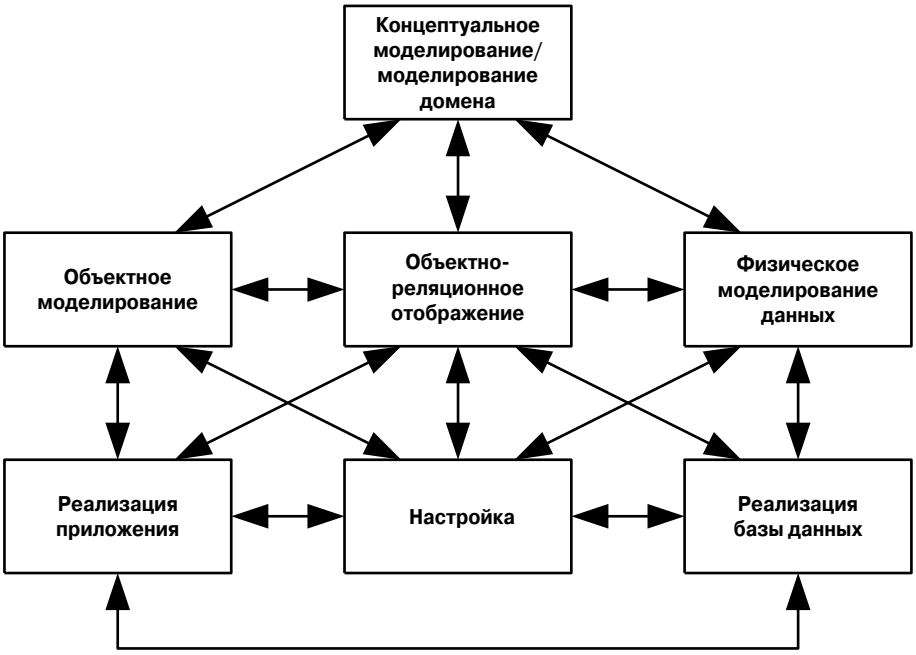

Рис. 2.6. Возможные направления деятельности при разработке эволюционного проекта

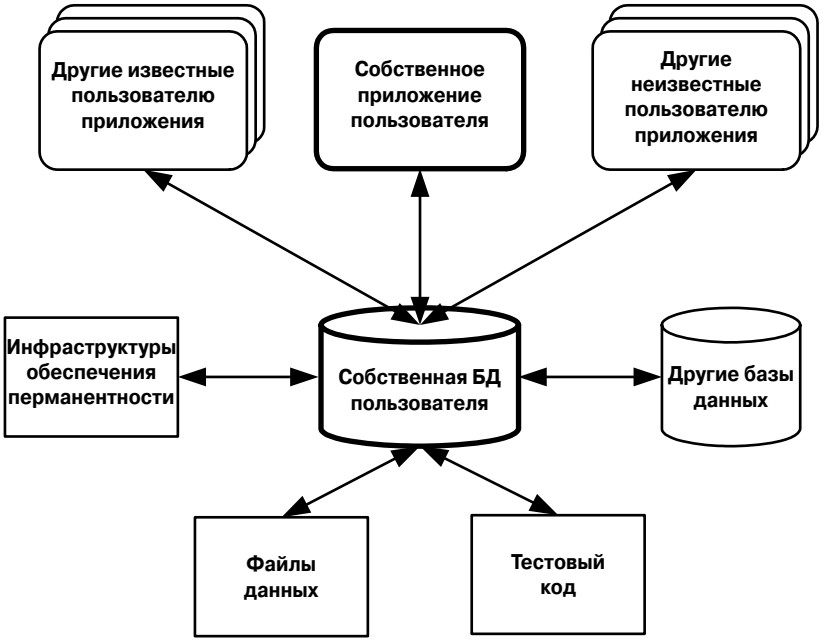

Рис. 2.7. Базы данных, тесно связанные с внешними программами

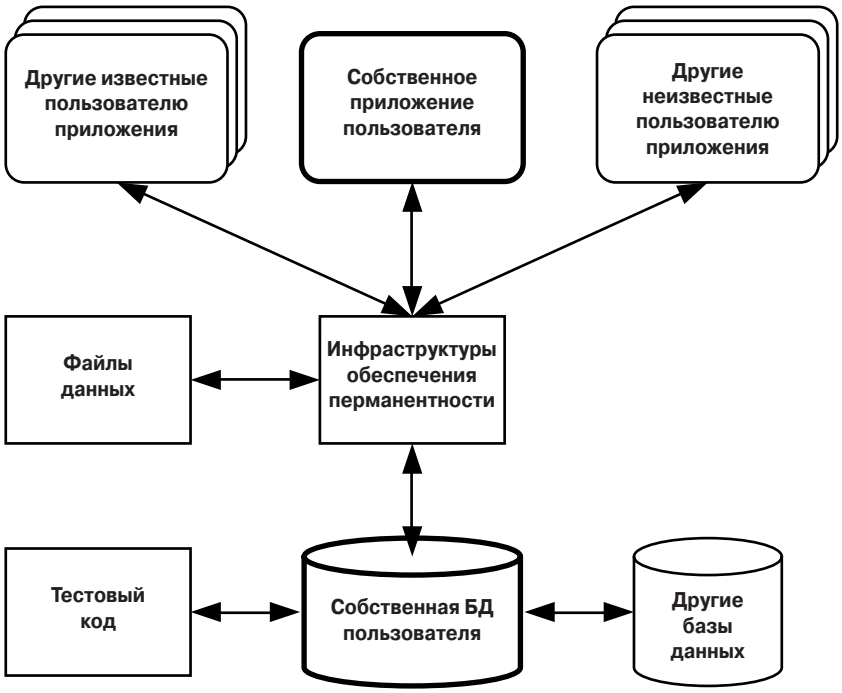

Рис. 2.8. Снижение степени связности путем создания дополнительного уровня доступа

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.