Глава 9

Проектирование программ и транзакций

Следует различать проектирование системы и проектирование программы. Проектирование программы относится к той области системного проектирования, которая связана с моделированием логики выполнения программы и определяет схему кооперативного взаимодействия объектов применительно к архитектуре клиент-сервер. Программы ИС выполняют бизнес-транзакции. Транзакция — это логический элемент базы данных, на который может оказать влияние СУБД при условии, что область действия транзакции определена в программе.

Проект программы и транзакции соединяет воедино артефакты системного проекта и знаменует собой кульминацию процесса системного проектирования. Он вносит логику приложения в проект GUI-интерфейса и базы данных. Результатом является проектный документ, который содержит достаточно подробные указания для того, чтобы программисты могли начать "кроить программу". На самом деле некоторая начальная программа может быть сгенерирована автоматически (прямое конструирование) на основе проекта. Расширение этой начальной программы, производимое программистами, можно подвергнуть "реконструкции", вновь вернувшись к проекту и завершая, таким образом, замкнутый цикл конструирования.

9.1. Проектирование программы

Проектирование программы — неотъемлемая часть проектирования всей системы (главы 6, 7 и 8). *Архитектурное проектирование* пакетов классов и компонент устанавливает исходную схему работы приложения. *Детализированное проектирование* GUI-интерфейса и баз данных определяет клиентскую и серверную составляющие этой схемы. Проектирование программы заполняет бреши в начальной схеме и превращает ее в проектный документ, который можно передать программистам для реализации.

Проектирование концентрируется одновременно на одной программе приложения. В этом смысле проектирование программы является прямым продолжением *проектирования пользовательского интерфейса*, рассмотренного в главе 7. При проектировании программы используется фрагмент (подсхема) *проекта базы данных* (глава 8) для

определения связанных с ним процедурных аспектов — хранимых процедур и определяемых программно триггеров.

Логика выполнения программы разделена между клиентским и серверным процессами. *Клиентский процесс* реализует большую часть динамики кооперативных взаимодействий объектов (разд.6.2), воплощенных в программе. Верный выбор соотношения между связностью и связанностью объектов (разд.9.1.1) может помочь справиться со сложностью этих взаимодействий. *Серверный процесс*, помимо прочего, следит за выполнением бизнес-транзакций, инициированных *клиентским процессом*.

9.1.1. Связность и увязка классов

Мимоходом, в частности в главах 5 и 6, мы определили основные принципы качественного проектирования программ, правда, по большей части в контексте качественного системного проектирования. Краеугольным камнем создания понятных, удобных в сопровождении и масштабируемых программ является введение *иерархии классов*. Чтобы избежать создания объектно-ориентированных программ, которые устаревают на следующий день после передачи их заказчикам, необходимо правильно пользоваться такими мощными (в том числе по своей разрушительной силе!) средствами, как *наследование* и *делегирование*.

Надлежащее проектирование программы предполагает сбалансированность между *связностью* (*cohesion*) и *увязкой* (*coupling*) классов. Термины связность и увязка были выработаны в рамках методов структурного проектирования. Однако, эти термины имеют похожий смысл и значение и для объектно-ориентированного проектирования [60], [77].

Под *связностью классов* понимают степень внутреннего самоопределения класса. Связность является мерой независимости класса. Классы, обладающие сильной связностью, выполняют одно действие или их функционирование подчинено одной цели. Чем выше уровень связности, тем лучше.

Под *увязкой* классов понимают, насколько тесно связаны между собой классы. Увязка является мерой взаимозависимости классов. Чем ниже уровень увязки, тем лучше (однако, не следует забывать, что для кооперации классы должны быть “увязаны”!).

Связность и увязка находятся в противоречии друг с другом. Лучшая связность приводит к ухудшению увязки и наоборот. Задачей конструктора является достижение их наилучшей сбалансированности. Риль (Riel) [70] предложил несколько эвристик, посвященных этой проблеме.

- Два класса должны быть либо независимы друг от друга, либо один из классов должен зависеть только от открытого интерфейса другого класса.
- Атрибуты и связанные методы должны находиться в одном классе (эта эвристика часто нарушается классами, обладающими большим количеством методов *открытия доступа* (get, set), определенных в их открытом интерфейсе).
- Класс должен охватывать одну и только одну абстракцию. Информация, не имеющая отношения к данной абстракции, когда подмножество методов работает на соответствующем подмножестве атрибутов, должна быть перенесена в другой класс.
- Системная логика должна быть распределена по возможности равномерно (так, чтобы классы были равномерно загружены совместной работой).

9.1.1.1. Виды увязки классов

Чтобы взаимодействовать, классы должны быть “увязаны”. Между классом X и классом Y существует увязка, если класс X может непосредственно ссылаться на класс Y. Пейдж-Джонс (Page-Jones) [60] приводит восемь типов увязки классов (которые он называет набором непосредственных межклассовых ссылок).

1. X наследует от Y.
2. X обладает атрибутом класса Y.
3. X обладает атрибутом шаблона с параметром класса Y.
4. X обладает методом со входным аргументом класса Y.
5. X обладает методом с выходным аргументом класса Y.
6. X осведомлен о глобальной переменной класса Y.
7. X осведомлен о методе, содержащем локальную переменную класса Y.
8. X — класс, дружественный классу Y.

9.1.1.2. Закон Деметра

Увязка классов необходима для взаимодействия объектов, однако, как мы заметили в разд. 5.2, она должна быть по возможности ограничена рамками уровней иерархии классов (т.е. сведена к увязке внутри уровня). Межуровневое взаимодействие должно быть сведено к минимуму и тщательно направляться. Дополнительные руководящие принципы для ограничения произвольного взаимодействия классов сформулированы в законе Деметра (Demeter) [50].

Закон Деметра определяет, на что могут быть нацелены сообщения в рамках методов класса. Он гласит, что целью сообщений может быть только один из перечисленных ниже объектов [60].

1. Объект, в котором определен метод (т.е. в C++ — это `this`, в Java — `self`, а в Smalltalk — `super`).
2. Объект, который является аргументом сигнатуры метода.
3. Объект, на который ссылается атрибут объекта (включая объект, на который ссылается один из атрибутов коллекции).
4. Объект, созданный методом.
5. Объект, на который ссылается глобальная переменная.

Для ограничения увязки, вызванной наследованием, можно ограничить третье правило атрибутами, определенными в самом классе. Тогда атрибут, унаследованный классом, не может использоваться для обозначения целевого объекта для сообщения. Это ограничение известно как *строгое правило Деметра* [60].

9.1.1.3. Методы открытия доступа и бессмысленные классы

Как упоминалось в разд. 9.1.1, атрибуты и связанные с ними методы должны находиться в одном классе [70]. Класс должен сам “решить свою судьбу”. Он может ограничить доступ других классов к своему состоянию за счет запрещения методов открытия доступа в своем интерфейсе. *Методы открытия доступа* (*accessor methods*) определяют операции *наблюдателя* (*observer* (`get`)) или *мутатора* (*mutator* (`set`)) (разд.8.3.1.2).

Методы открытия доступа “открывают” класс для внутренних манипуляций со стороны другого класса. Хотя увязка и подразумевает определенную долю использования объекта другого класса “в своих интересах”, чрезмерная распространенность методов открытия доступа может привести к неравномерному распределению “интеллектуальной” нагрузки на классы. Класс, содержащий слишком много методов, рискует оказаться *бессмысленным* — другие классы определяют, что для него “хорошо”.

В свете сказанного становится ясно, что существуют ситуации, когда класс должен “открыться” другим классам. Это имеет место всякий раз, когда необходимо реализовать некоторую политику (или стратегию) “отношений” между двумя или более классами [70]. За примером недалеко ходить.

Предположим, что у нас имеется два класса `INTEGER` и `REAL` и нам необходимо реализовать “политику” преобразования целых (integer) чисел в вещественные (real) и наоборот. Для этого требуется ответить на некоторые вопросы. В каком из двух классов должна быть реализована политика? Требуется ли нам специальный класс-преобразователь `CONVERTER` для реализации этой политики? В противном случае, по крайней мере, один из классов должен допускать использование методов открытия доступа и вследствие этого может стать “бессмысленным” по отношению к этой политике.

Здесь уместно будет привести знаменитое высказывание Пейдж-Джонса: “На объектно-ориентированной ферме и молоко объектно-ориентированное. Значит ли это, что объектно-ориентированная корова должна отправлять объектно-ориентированному молоку сообщение `uncow_yourself` (“освободиться от коровы”), или объектно-ориентированное молоко должно отправлять объектно-ориентированной корове сообщение `unmilk_yourself` (“избавиться от молока”)” (из выступления Пейдж-Джонса на конференции OOPSLA'87).

Пример 9.1. Запись на университетские курсы

В главе 8 мы разработали схему кооперации для части приложения “Запись на университетские курсы” (примеры 6.3 и 6.4 в разд. 6.2.4.1 и 6.2.4.2). Модели кооперации обеспечивают относительно равномерное распределение функциональной нагрузки, однако, при этом никакие альтернативные решения не рассматривались.

Предположим, к примеру, что нам необходимо добавить студента в список слушателей курса. Для этого следует осуществить две проверки. Во-первых, необходимо выявить, изучение каких курсов является обязательным условием для прослушивания данного курса. Во-вторых, необходимо проверить учебное личное дело студента, чтобы выяснить, удовлетворяет ли студент этим обязательным условиям. Зная это, мы можем принять решение, можно ли студента внести в число слушателей курса.

Предположим, что сообщение `enrol()` (записать [на предлагаемый курс]) должно отправляться пограничному объекту `:EnrolmentWindow`. Предположим также, что три класса — `CourseOffering`, `Course`, и `Student` (Предлагаемый курс, Курс и Студент) кооперируются для выполнения задачи. Наша задача заключается в том, чтобы выработать круг возможных диаграмм кооперации для решения проблемы. Рассмотрите аргументы за и против различных вариантов решения.

На рис. 9.1 показан первый вариант решения. Пограничный объект `:EnrolmentWindow` инициирует транзакцию с помощью отправки сообщения `enroll()` объекту `aCourse`. Объект `aCourse` запрашивает у объекта `aStudent` данные относительно учебного досье и сравнивает их с обязательными условиями. Объект `aCourse` решает, может ли объект `aStudent` быть внесен в список, и дает указание объекту `aCourseOffering` добавить `aStudent` в его список студентов.

Сценарий, показанный на рис. 9.1, дает слишком много полномочий объекту aCourse. Объект aCourse становится, так сказать, "вершителем политики". В этих условиях роль объекта aStudent лишена смысла. Решение не сбалансировано, однако, ясного выхода из ситуации не видно.

Можно переместить акценты с объекта aCourse на объект aStudent, в результате получим решение, показанное на рис. 9.2. Теперь пограничный объект :EnrolmentWindow требует от объекта aStudent выполнить основную работу. Объект aStudent вызывает метод-наблюдатель getPrereq() объекта aCourse. Объект aStudent решает, возможна ли запись, и дает указание объекту aCourseOffering записать студента.

Рис. 9.3 является иллюстрацией более сбалансированного решения, при котором вершителем политики является объект aCourseOffering. Решение "беспристрастно" по отношению к объектам aCourse и aStudent, но оно делает существование этих двух объектов довольно бесполезным и бессмысленным. Объект aCourseOffering действует подобно "главной программе" (по выражению Риеля (Riel) [70], играет роль *класса "Бог"*).

Решение, представленное на рис. 9.3, можно улучшить за счет введения *управляющего объекта*, призванного обрабатывать "политическую информацию" (см. подход *BCE* — разд.5.2.4). Управляющий объект :EnrolmentPolicy на рис. 9.4 устраняет увязку трех классов-сущностей из политики записи на курсы. Это сулит определенные преимущества, так как любые изменения политики инкапсулированы в пределах одного управляющего класса. Однако, существует риск, что класс EnrolmentPolicy может вырасти в класс "Бог".

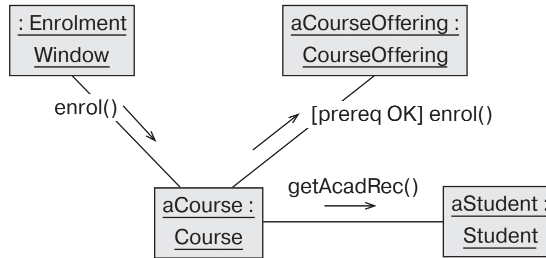

Рис. 9.1. Объект aCourse в качестве "вершителя политики" (Запись на университетские курсы)

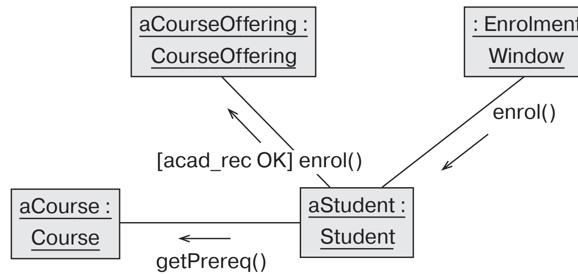

Рис. 9.2. Объект aStudent в качестве "вершителя политики" (Запись на университетские курсы)

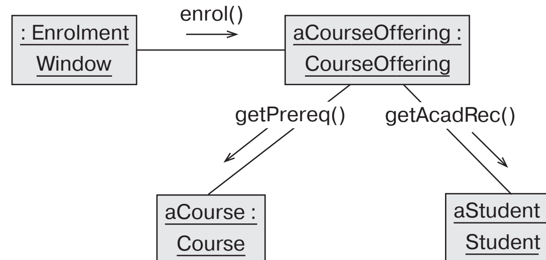

Рис. 9.3. Объект aCourseOffering в качестве "вершителя политики" (Запись на университетские курсы)

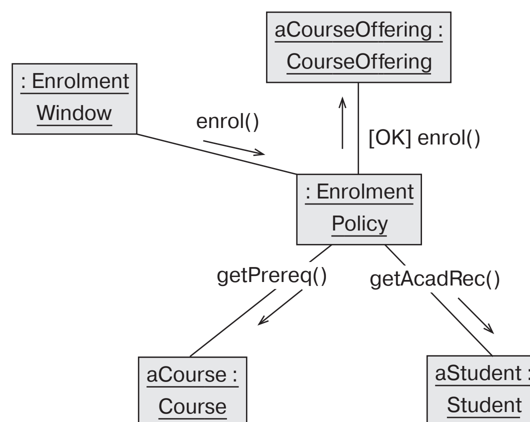

Рис. 9.4. Класс: EnrolmentPolicy в качестве "вершителя политики" (Запись на университетские курсы)

9.1.1.4. Динамическая классификация и связность классов со смешанными экземплярами

В разд.2.1.5.2.3 мы поднимали вопрос относительно *динамической классификации* и заметили, что наиболее распространенные объектно-ориентированные среды программирования ее не поддерживают. Цена, которую приходится платить за отсутствие подобной поддержки, зачастую выражается в проектировании классов, которые отличаются связностью смешанных экземпляров.

Пейдж-Джонс дает следующее определение подобного класса: "Класс со *связностью смешанных экземпляров* (*mixed-instance cohesion*) обладает некоторыми свойствами, которые не определены для некоторых объектов класса". Некоторые методы класса применимы только к подмножеству объектов этого класса, и некоторые атрибуты имеют смысл только для подмножества объектов.

Например, класс `Employee` может определять объекты, которые являются как "обычными" работниками, так и менеджерами. Менеджерам выплачивается надбавка. Отправка сообщения о выплате надбавки `payAllowance` объекту `Employee` не имеет смысла, если этот объект `Employee` не принадлежит к подмножеству объектов-менеджеров.

Для того, чтобы избавиться от связности смешанных объектов, не требуется расширять иерархию обобщения для выделения подклассов `Employee`, таких как `OrdinaryEmployee` и `Manager`. Однако, объект `Employee` может в один день принадлежать категории обычных работников `OrdinaryEmployee`, а в другой день перейти в категорию менеджеров `Manager` или наоборот. Для того, чтобы исключить связность смешанных объектов, нам требуется разрешить объектам динамически изменять принадлежность классу во время выполнения программы — как говорится, "опять двадцать пять", если динамическая классификация не поддерживается.

Пример 9.2. Запись на университетские курсы

Рассмотрите следующие варианты примера 9.1.

- Посещение вечернего курса возможно только для студентов-вечерников.
- Студенты стационарного обучения могут записываться только на дневные курсы.
- Если студенты-вечерники желают записаться на вечерние курсы, они должны внести небольшую дополнительную плату.
- Студенты-вечерники автоматически рассматриваются как студенты стационара, если они записываются более, чем на шесть кредитных пунктов (т.е. обычно более, чем на два предложенных курса) в данном семестре (и наоборот).

Наша задача заключается в том, чтобы предложить сильно связную структурную модель кооперации без связности смешанных экземпляров. Затем предложенную модель следует подвергнуть критической оценке и выдвинуть и обсудить альтернативное решение, позволяющее избежать проблем с динамической классификацией.

Чтобы исключить связность смешанных экземпляров, требуется произвести специализацию класса `Student` в виде двух подклассов `PartTimeStudent` (Студент-заочник) и `FullTimeStudent` (Студент стационара) (рис. 9.5). Если каждый студент должен быть отнесен к категории студентов вечерней или дневной формы обучения, класс `Student` является абстрактным классом. Сообщение о необходимости внести дополнительную плату `payExtraFee(crs_off)` никогда не отправляется объекту класса `FullTimeStudent`, поскольку `FullTimeStudent` не содержит методов его обработки.

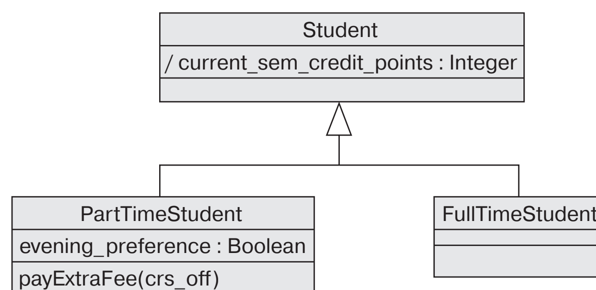

Рис. 9.5. Структурная схема кооперации, исключающая связность смешанных экземпляров (Запись на университетские курсы)

Следует согласиться, что у нас по-прежнему остались проблемы. Студент-вечерник может отдать предпочтение изучению дневного курса (т.е. `evening_preference = 'False'`), и при этом никакая дополнительная плата не вносится. Другими словами,

для подкласса PartTimeStudent мы по-прежнему сталкиваемся с проблемой связности смешанных экземпляров. Отправка сообщения payExtraFee(crs_off) объекту PartTimeStudent не имеет смысла, если студент берет дневной предлагаемый курс.

На рис. 9.6 показано расширение схемы кооперации, позволяющее исключить второй аспект связности смешанных экземпляров. Для этого введен новый специализированный класс, отражающий существование студентов-вечерников, отдающих предпочтение прослушиванию дневных курсов обучения. Класс DayPrefPartTimeStudent не обладает методом payExtraFee(crs_off). Однако, что делать, если объект DayPrefPartTimeStudent вынужден "брать" предложение вечернего курса, поскольку на дневном курсе обучения нет свободных мест? Наверное, в этом случае применим какой-то другой вид дополнительной оплаты. Должны ли мы продолжать развивать нашу специализацию классов, чтобы дойти до производного класса, отражающего существование студентов-вечерников, отдающих предпочтение прослушиванию дневных курсов обучения, которым не удается попасть на эти курсы — UnluckyDayPrefPartTimeStudent?

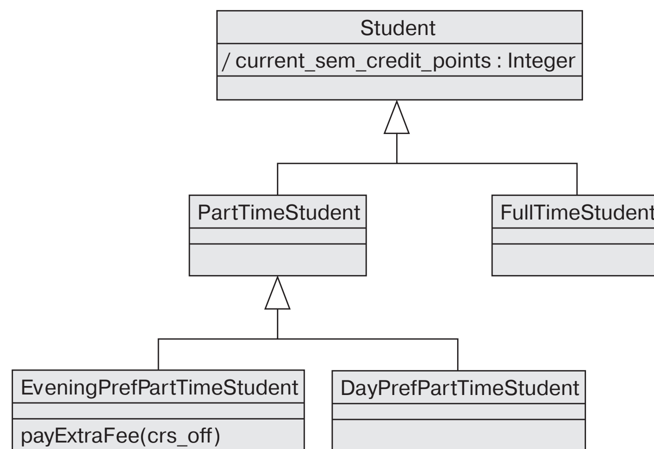

Рис. 9.6. Расширенная структурная схема кооперации, исключающая еще один аспект связности смешанных экземпляров (Запись на университетские курсы)

Чтобы не оказаться в нелепом положении, следует отказаться от идеи все дальше и дальше продолжать процесс исключения связности смешанных экземпляров. И больше мы не станем упоминать динамическую классификацию... В действительности, текущее значение атрибута current_sem_credit_points определяет, является ли студент студентом-вечерником или студентом дневной формы обучения.

Аналогично, студент может в любой момент изменить свое предпочтение в отношении вечернего или дневного времени посещения предложенного курса. В отсутствие поддержки *динамической классификации* со стороны среды программирования обеспечение возможности для объекта изменить принадлежность классу во время исполнения программы является прерогативой программиста. Это грубый метод — очень грубый в случае постоянных объектов, значения OID которых содержат идентификаторы класса.

Альтернативное решение состоит в том, чтобы ограничить глубину иерархии наследования, исключить необходимость динамической классификации и вновь возвратиться к наличию определенного уровня связности смешанных экземпляров. Например, мы можем настоять на использовании структурной кооперации, схема которой приведена на рис. 9.5, и решить проблему, связанную с предпочтениями прослушивания курса в вечернее время, позволив объекту по-разному реагировать на сообщение `payExtraFee(crs_off)` в зависимости от значения атрибута `evening_preference`. Подобное решение можно запрограммировать с помощью оператора `if`, как показано в приведенном ниже примере псевдокода.

```
method payExtraFee(crs_off) for the class PartTimeStudent
      if evening_preference = ФFalseХ
          return
      else
          do it
end method
```

Хотя использование оператора if в объектно-ориентированной программе означает отказ от наследования и полиморфизма, это может оказаться неизбежным по чисто прагматическим соображениям. Вместо того, чтобы бороться с *динамической классификацией*, программист вводит в класс *динамическую семантику*. Объект по-разному реагирует на одно и то же сообщение в зависимости от его локального текущего состояния. Для проектирования динамической семантики для класса можно использовать диаграммы состояний. В ходе проектирования, вполне вероятно, пострадает *связность* класса.

9.1.2. Проектирование клиент-серверных кооперативных взаимодействий

Чтобы получить доступ к данным программы, ИС взаимодействуют с базами данных. Для доступа к данным в базе данных и их модификации клиентская программа должна использовать язык баз данных — обычно SQL. Чтобы понять, каким образом клиентская программа связывается с сервером баз данных, требуется признать существование целого ряда диалектов языка SQL, которые, кроме того, можно использовать на различных уровнях программной абстракции.

На рис. 9.7 показаны пять различных уровней SQL-интефейса. На *уровне 1* SQL используется как язык определения данных (data definition language — DDL). DDL — язык спецификации для определения структур баз данных (схем баз данных). Основными пользователями языка SQL являются проектировщик и администратор баз данных (database administrator — DBA).

На *уровне 2* SQL используется как язык манипулирования данными (data manipulation language —DML) или *язык запросов* (*query language*). Термин язык запросов, однако, неверен, поскольку SQL на уровне 2 служит не только цели доступа к данным, но и их модификации (включая операции вставки, обновления и удаления).

SQL уровня 2 применяет широкий круг пользователей, начиная с неискушенных случайных пользователей и заканчивая опытными DBA. На этом уровне SQL представляет собой язык *интерактивного взаимодействия*, а это означает, что пользователь может сформировать запрос вне какой-либо среды программирования и немедленно

выполнить его над базой данных. SQL уровня 2 является отправной точкой в изучении более развитых SQL следующих уровней.

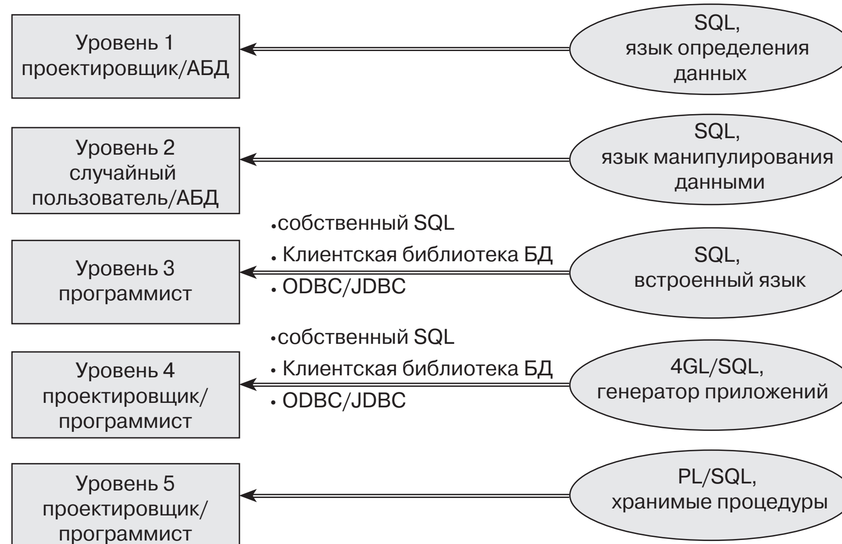

Рис. 9.7. Классификация уровней SQL-интерфейса

Прикладные программисты используют SQL уровней выше 2-го. На этих более высоких уровнях SQL позволяет работать в режиме *последовательной обработки записей* (*record-at-a-time processing*) в дополнение к возможностям *последовательной обработки наборов записей* (*set-at-a-time processing*) (единственно доступных) на уровне 2. При работе в режиме последовательной обработки наборов записей СУБД берет в качестве входа для запроса одну или более таблиц (набор записей) и возвращает в качестве выхода таблицу. Хотя это довольно мощное средство, использовать его в сложных запросах не только трудно, но и небезопасно.

Чтобы быть уверенным, что запрос возвращает верный результат, программист должен иметь возможность просматривать одну за другой записи, возвращаемые как результат запроса, и принимать решение о том, что делать с каждой отдельной записью. Подобная возможность последовательной обработки записей, которая называется *курсором* (*cursor*), доступна в SQL уровней выше 2-го.

SQL *уровня 3 встроен* в традиционный язык программирования, наподобие C или Cobol. Поскольку компилятор языка программирования не понимает SQL, необходим предкомпилятор (препроцессор) для трансляции SQL-операторов в вызовы функций DB-библиотеки, поставляемой изготовителем СУБД. Программист может предпочесть программу, непосредственно использующую функции DB-библиотеки, при этом необходимость в предкомпиляторе отпадает.

Одним из самых распространенных способов обеспечения интерфейса клиентской программы с базами данных является использование стандартов *ODBC* (Open Database Connectivity — открытый интерфейс доступа к базам данных) или *JDBC* (Java Database Connectivity — интерфейс организации доступа Java-приложений к базам данных). Для программирования с использованием этих интерфейсов требуется

программный драйвер ODBC или JDBC для определенной СУБД. Интерфейсы ODBC и JDBC обеспечивают стандартную надстройку над языком SQL, которая с помощью драйвера транслируется в "родной" SQL СУБД.

Интерфейсы ODBC/JDBC обладают тем преимуществом, что обеспечивают автономизацию программы от "родного" языка SQL СУБД. Если в будущем требуется перенесение программы на другую платформу СУБД, в качестве основного приема осуществления этого процесса служит простая замена драйверов. Что более важно, работа с интерфейсами ODBC/JDBC позволяет одному приложению выдавать запрос более, чем к одной СУБД,

Недостатком стандартов ODBC/JDBC является то, что они выступают по отношению к SQL "наименьшим общим знаменателем". Клиентское приложение не может воспользоваться никакими преимуществами специальных средств SQL или расширений, поддерживаемых поставщиком конкретной СУБД.

SQL *уровня 4* использует ту же стратегию встраивания SQL в клиентские программы, что и SQL уровня 3. Однако, SQL уровня 4 обеспечивает более мощную среду программирования на основе генератора приложений или графического языка четвертого поколения (fourth generation language — 4GL). Как правило, язык 4GL поставляется оснащенным средствами построения экранных изображений конструирования GUI-интерфейса. Поскольку приложения ИС требуют развитого GUI, для построения подобных приложений зачастую выбирают комплект 4GL/SQL.

SQL *уровня 5* дополняет SQL уровней 3 и 4, обеспечивая возможность переноса некоторых SQL-операторов из клиентской программы на активный (программируемый) сервер баз данных. SQL используется в качестве языка программирования (PL/SQL). Программы сервера можно вызвать изнутри клиентской программы, как рассматривается далее.

9.1.2.1. Хранимые процедуры

Хранимые процедуры (*stored procedure*) впервые были введены в СУБД Sybase и сейчас составляют неотъемлемую часть всех основных коммерческих СУБД. Хранимые процедуры превращают базу данных в активную программируемую систему.

Хранимая процедура представляет собой расширение SQL, допускающее такие конструкции, как переменные, циклы, ветвления и операторы присваивания. Хранимые процедуры получают имя, могут принимать входные и выходные параметры, они компилируются и хранятся в базе данных. Клиентская программа может вызвать хранимую процедуру во многом аналогично тому, как она вызывает подпрограмму.

Рис. 9.8 является иллюстрацией преимуществ вызова клиентской программой хранимой процедуры вместо отправки полного запроса серверу. Запрос, сформированный в клиентской программе, пересылается серверу по сети. Запрос может содержать синтаксические ошибки и ошибки других типов, но клиент не в состоянии исключить их — единственное место, где можно осуществить подобную верификацию, — система базы данных. После верификации СУБД проверяет, обладает ли вызывающая сторона правами, достаточными для выполнения запроса. Если это так, то запрос оптимизируется для определения наилучшего плана доступа к данным. Только после этого запрос компилируется, выполняется, и результат возвращается клиенту.

С другой стороны, если запрос (или весь набор запросов) написан в виде хранимой процедуры, он оптимизируется и компилируется с последующим сохранением на сервере баз данных. Клиентской программе нет необходимости пересылать (воз-

можно, довольно объемный) запрос по сети — вместо этого она отправляет короткий вызов, содержащий имя процедуры и список фактических параметров. Если повезет, процедура может располагаться в кэш-памяти СУБД. Если нет, она переносится в память из базы данных.

Полномочия пользователя внимательно анализируются так же, как и в случае SQL-запроса. Все фактические параметры подставляются вместо формальных параметров, и хранимая процедура выполняется. Результат возвращается вызывающей программе.

Как видно из приведенного выше сценария, хранимые процедуры дают значительно более эффективный способ доступа к базе данных из клиентской программы. Преимущества в производительности являются следствием экономии *сетевого трафика* и отсутствия необходимости *синтаксического разбора и компиляции* всякий раз при получении клиентского запроса. Что еще более важно, *сопровождение хранимой процедуры локализовано в единственном месте*, а вызываться она может из многих клиентских программ.

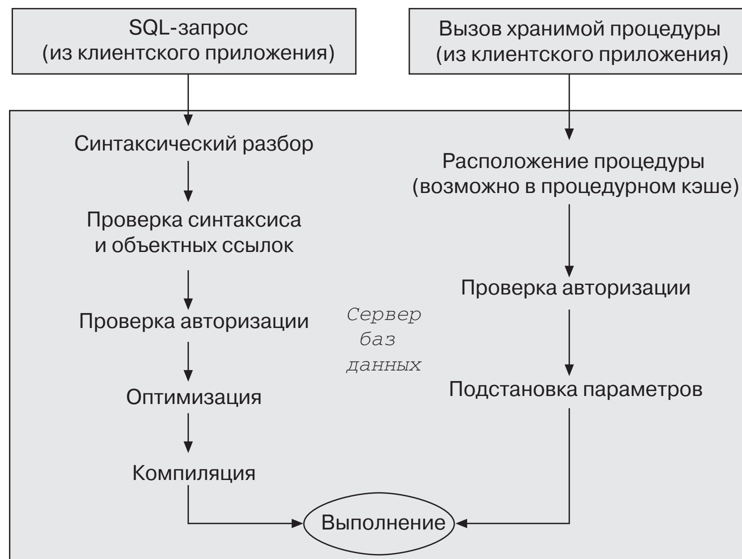

Рис. 9.8. Сравнение вызова из клиентской программы SQL-запроса и хранимой процедуры

9.1.2.2. Триггеры

Триггеры (разд.8.4.1.4) представляют собой особый вид хранимой процедуры, которую нельзя вызвать, — они либо запускают себя, либо добавляют (`insert`), обновляют (`update`) или удаляют (`delete`) события из таблицы. Это значит, что каждой таблице можно сопоставить до трех триггеров. Конечно, в некоторых системах это так и есть (например, Sybase). В других системах (например, Oracle) выделяются дополнительные варианты событий, что приводит к возможности держать для каждой таблицы более трех триггеров. (Тем не менее, доступность большего количества типов триггеров не придает большей выразительной силы триггерным программам).

Триггеры можно программировать для введения некоторых *бизнес-правил*, которые применяются к базе данных и не могут быть нарушены ни одной клиентской про-

граммой или интерактивным DML-оператором языка SQL. Это означает, что триггеры можно использовать поверх процедурного способа обеспечения ограничений ссылочной целостности (как было рассмотрено в разд.8.4.1.4). Например, можно создать триггер, который запретит доступ к базе данных в течение определенных периодов времени или в некоторые дни.

Пользователь клиентского приложения может даже не знать о том, что триггеры "ожидают", когда в базе данных произойдут какие-либо модификации. Если модификации не нарушают бизнес-правил, триггеры невидимы для программ. Триггер "выказывает" себя пользователю при попытке выполнить недопустимую DML-команду. Он оповещает пользователя о проблеме, отображая информационное сообщение на экране программы и отказываясь выполнить DML-операцию.

9.2. Навигация по программе

При наличии активной базы данных количество программных объектов возрастает, и кооперативное взаимодействие объектов становится более сложным. Диаграмм навигации по окнам, используемых в качестве проектного документа, больше недостаточно для того, чтобы программист мог взяться за реализацию. Их необходимо расширить (включить в) до более полных диаграмм навигации по программе.

В рамках UML не существует какой-либо стандартизованной концепции в отношении *программной навигации.* Тем не менее, это основополагающая абстракция моделирования, необходимая для преодоления разрыва между проектом и реализацией системы. Альтернативой может быть только плохая инженерия программных средств — *архитектура и логика* программы не документированы и отдаются на откуп программистам.

9.2.1. Стереотипы диаграммы видов деятельности для навигации по программе

Для преобразования диаграммы навигации по окнам в диаграмму навигации по программе требуется добавить серверные стереотипы к диаграммам видов деятельности UML. Необходимо ввести стереотипы как для *состояний* (прямоугольники со скругленными углами), так и для *видов деятельности* (овалы). Стереотипы должны учитывать особенности модели СУБД или даже особенности конкретной СУБД. (Это может служить основной причиной или, скорее, оправдывает отсутствие диаграмм навигации в UML).

Для проектирования диаграмм навигации по программам применительно к РСУБД можно использовать приводимый ниже перечень стереотипов. В зависимости от уровня абстракции, на котором должны быть сконструированы диаграммы навигации, перечень может сократиться или, наоборот, расшириться за счет дополнительных объектов.

- *Состояния* (*объекты данных*)
    - База данных
    - Реляционная таблица
  - Столбец
  - Запись
    - Реляционное представление

- Столбец
- Запись
  - Индекс
  - Кластер
- *Виды деятельности* (*программные объекты*)
  - Хранимая процедура
  - Триггер
    - Триггер типа On Insert
    - Триггер типа On Update
    - Триггер типа On Delete
    - Другие типы триггеров ...
  - Клиентский SQL-запрос
    - Запрос с использованием собственных средств СУБД
    - Запрос с использованием DB-библиотеки
    - Запрос с использованием ODBC/JDBC

9.2.2. Диаграмма навигации по программе

Чтобы проиллюстрировать диаграммы навигации по программе, мы просто расширим примеры диаграмм навигации по окнам, представленные в разд. 7.6.2. На рис. 9.9 показана диаграмма навигации по программе, которая расширяет (и несколько переупорядочивает) диаграмму навигации по окнам, представленную на рис. 7.23.

На рис. 9.9 введены два состояния сервера: `Inventory Control` (`<<database>>`) и `Product` (`<<table>>`). Добавлены три вида деятельности сервера: `Select Product` (`<<stored procedure>>`), `Update Product` (`<<stored procedure>>`) и `On Update Product` (`<<trigger>>`). Мы также добавили клиентский вид деятельности `Scroll` (`<<scrollbar/keyboard>>`).

Вместе с добавленной информацией диаграмма навигации по программе, представленная на рис. 9.9, информирует программиста о том, что для отображения товаров в окне `Product Browser` необходимо вызвать хранимую процедуру (`Select Product`). Эта же хранимая процедура задействуется, когда пользователь осуществляет прокрутку окна броузера вверх и вниз и информационное наполнение окна должно обновляться из базы данных.

Пример 9.4. Управление контактами с клиентами

Рассмотрите следующие расширения примера 7.5 (разд.7.6.2).

- Хранимые процедуры используются для модификации мероприятий (операции Insert, Delete, Update, Complete).
- Для отслеживания операций Insert и Update над таблицей мероприятий `Event` используются триггеры.

Наша задача заключается в проектировании диаграммы навигации по программе для той части приложения "Управление контактами с клиентами", которая обрабатывает модификацию мероприятий.

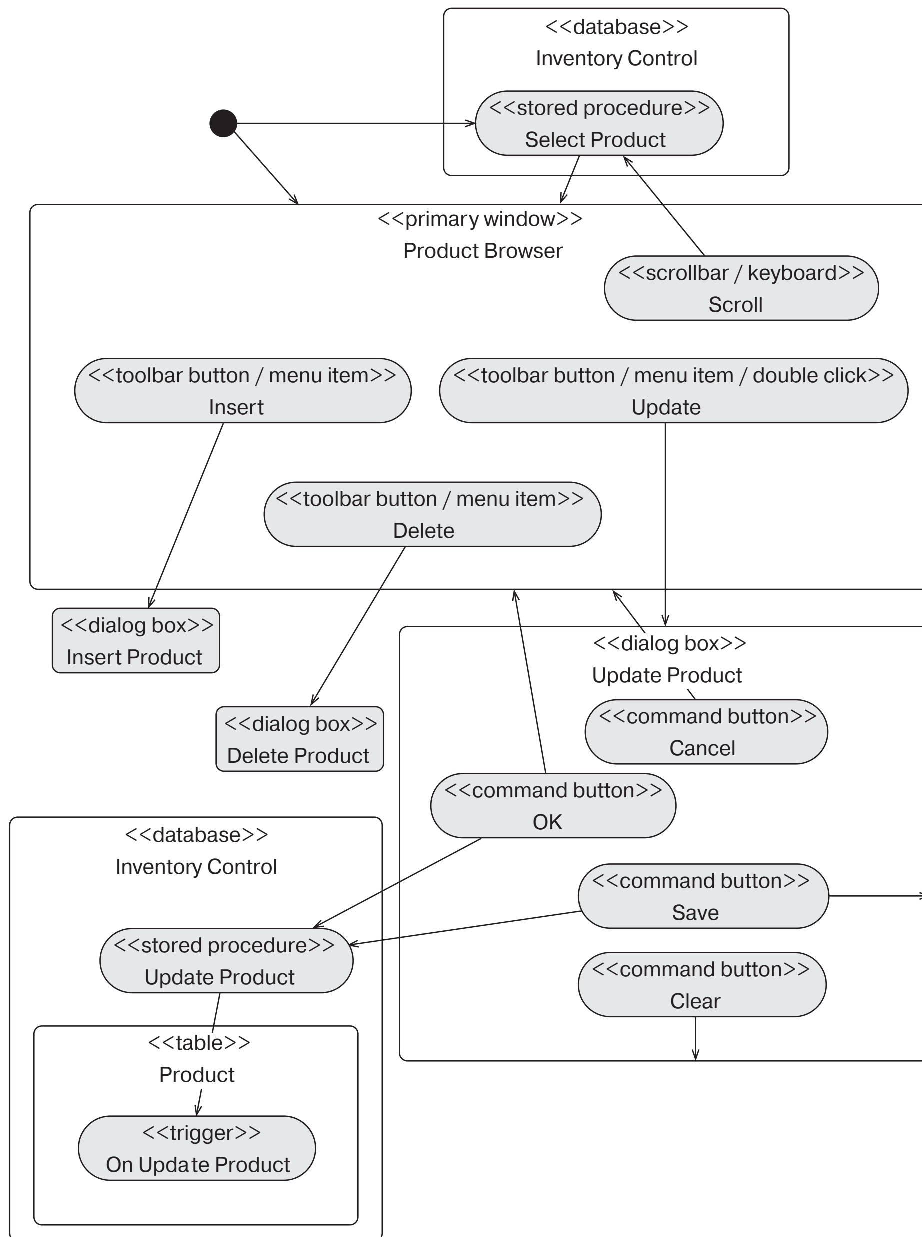

Рис. 9.9. Диаграмма навигации по программе

В левой части диаграммы можно увидеть, что нажатие кнопки OK или Save в диалоговом окне `Update Product` (`<<dialog box>>`) вызывает хранимую процедуру `Update Product` (`<<stored procedure>>`). Триггер контролирует результат модификации данных о товарах (`On Update Product`).

Наше решение для примера (рис. 9.10) включает только те состояния и виды деятельности клиентской части приложения с рис. 7.24, которые касаются модификации мероприятий. Заметим, что хранимую процедуру завершения мероприятия `Complete Event` можно вызвать либо из главного окна, либо из диалогового окна. Поскольку процедура `Complete Event` обновляет таблицу `Event`, срабатывает триггер `On Update Event`. Тот же триггер может быть активизирован хранимой процедурой сохранения информации по мероприятию `Save Event`.

Диалоговое окно `Task/Event Details` служит двойной цели введения информации по новому событию и обновления информации по существующему мероприятию. Командная кнопка `OK` вызывает хранимую процедуру `Save Event`. Список параметров вызова сообщает процедуре, означает ли нажатие кнопки `OK` операцию вставки или обновления. Соответственно, выполнение процедуры вызывает срабатывание одного из двух триггеров: `On Insert Event` или `On Update Event`.

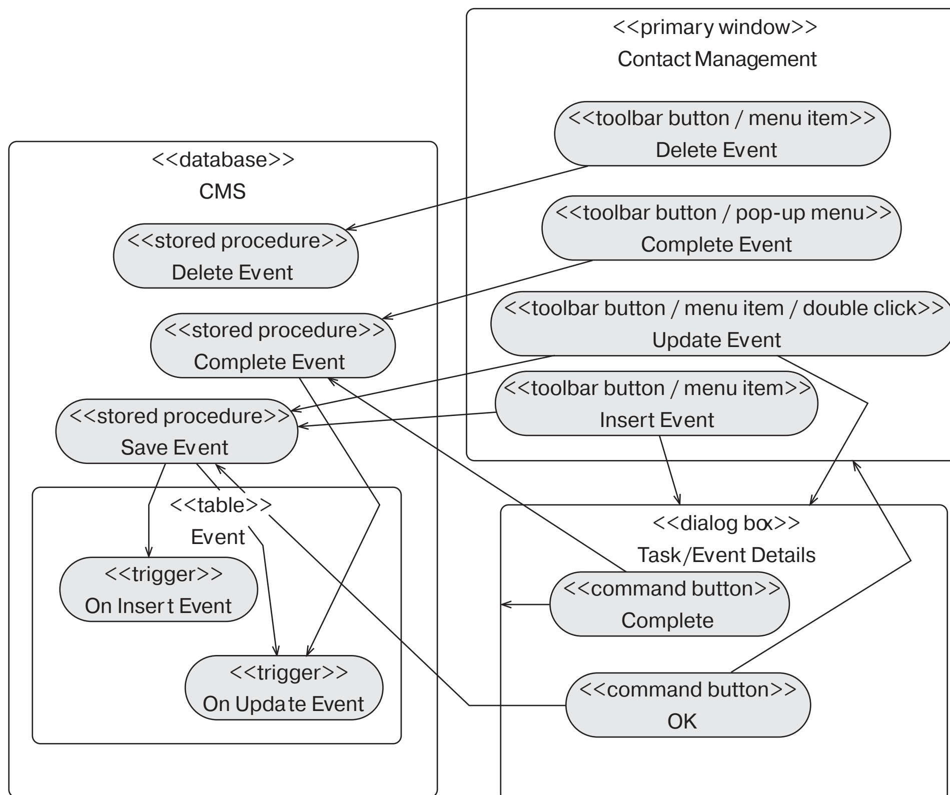

Рис. 9.10. Диаграмма навигации по программе (Управление контактами с клиентами)

9.3. Проектирование транзакций

Транзакция — это логическая единица работы, которая состоит из одного или более операторов SQL, выполняемых пользователем. Транзакция также является единицей измерения *непротиворечивости базы данных* — состояние базы данных после завершения

транзакции непротиворечиво. Для обеспечения этой непротиворечивости используется программа менеджера транзакций, которая служит двум целям: *восстановлению базы данных* (*database recovery*) и *управлению параллельным выполнением операций* (*concurrency control*). (Последний вид управления называется также контролем совпадений или контролем противоречивых запросов к распределенной базе данных. *Прим. ред.*).

Согласно стандартам SQL транзакция начинается с первого исполняемого SQL-оператора (в некоторых системах требуется явный оператор начала транзакции наподобие `begin transaction`). Транзакция завершается операторами `commit` или `rollback`. Оператор `commit` записывает изменения в базу данных как постоянные. Оператор `rollback` стирает любые изменения, произведенные транзакцией.

Транзакция отличается свойством *атомарности* (*atomic*) — результат всех SQL-операторов, содержащихся в транзакции, либо полностью подтверждается (`commit`), либо отвергается (`rollback`). Пользователь определяет продолжительность (длину) транзакции. В зависимости от характера бизнес-требований, области приложения, стиля взаимодействия пользователя с компьютером транзакция может быть совсем короткой, состоящей из одного SQL-оператора, или содержать последовательность SQL-операторов.

9.3.1. Короткие транзакции

Для большинства традиционных приложений ИС требуются короткие транзакции. Короткая транзакция содержит один или более SQL-операторов, которые должны быть выполнены как можно быстрее, так, чтобы не задерживать другие транзакции.

Рассмотрим систему бронирования мест авиакомпании. Несколько туристических агентств принимают заказы от путешественников на билеты для полетов рейсами этой авиакомпании по всему миру. Существенным моментом здесь является быстрое выполнение СУБД каждой транзакции заказа, так чтобы постоянно иметь доступ к обновленной информации о наличии мест и база данных была готова к обработке следующей транзакции, ожидающей в очереди.

9.3.1.1. Пессимистическое управление параллельностью

Архитектура традиционных СУБД — ОСУБД составляют здесь заметное исключение — ориентирована на короткие транзакции. Эти системы работают в соответствии с алгоритмом *пессимистического управления параллельностью* (*pessimistic concurrency control*). При обработке транзакцией каждого постоянного объекта она запрашивает *блокировку* (*lock*). Существует четыре типа блокировок по объектам.

1. *Привилегированная блокировка* (*exclusive lock*) (*по записи*) — другие транзакции должны ожидать, пока транзакция, удерживающая подобную блокировку, завершится и освободит блокировку.
2. *Блокировка обновления* (*update lock*) (*по возможной записи*) — другие транзакции могут читать объект, однако, транзакции, удерживающей блокировку, гарантируется возможность выполнить обновление в привилегированном режиме, как только у нее возникнет в этом потребность.
3. *Блокировка чтения* (*read lock*) (*с разделением*) — другие транзакции могут читать и, возможно, получить блокировку обновления по объекту.

4. *Отсутствие блокировки* (*no lock*) — другие транзакции могут обновлять объект в любой момент; подходит только для приложений, допускающих "*черновое чтение*", — т.е. транзакция зачитывает данные, которые могут быть модифицированы или даже удалены (другой транзакцией) до завершения транзакции.

9.3.1.2. Уровни изолированности

Описанным выше четырем типам блокировки соответствуют четыре *уровня изолированности* (*levels of isolation*) (или *уровня развязки*) между параллельно выполняющимися транзакциями. Решение о том, какой уровень изолированности подходит для смеси транзакций на базе данных, является прерогативой системного проектировщика. Ниже перечислены эти уровни [41].

1. *Возможно черновое чтение* — транзакция `t1` модифицировала объект, но еще не получила подтверждения; транзакция `t2` зачитывает объект; если `t1` осуществляет откат транзакции, то `t2` получает объект, который в известном смысле никогда не существовал в базе данных.
2. *Возможно неповторяемое чтение* — транзакция `t1` зачитала объект; `t2` обновляет объект; `t1` зачитывает тот же объект снова, но в этот момент она получает другое (по сравнению с первоначальным) значение для того же объекта.
3. *Возможно возникновение фантомного объекта* — транзакция `t1` зачитала набор объектов; `t2` вставляет новый объект в набор; `t1` повторяет операцию чтения и обнаруживает "объект-фантом".
4. *Повторяемое чтение* — транзакции `t1` и `t2` могут продолжать выполнение, но перемежающееся выполнение этих транзакций даст тот результат, что и их выполнение одна за другой (это называется *выполнением, поддающимся сериалиации* (*serializable execution*)).

Типичное интерактивное приложение ИС, основанное на использовании GUI-интерфейса, требует коротких транзакций. Однако, уровни изолированности различных транзакций в одном и том же приложении могут отличаться. Для этой цели можно использовать SQL-оператор `set transaction`. Сущность компромисса ясна — увеличение уровня изолированности ведет к общему снижению уровня параллелизма системы.

Одно из критически важных проектных решений, однако, не зависит от рассмотренных выше вопросов. Начало транзакции всегда должно откладываться до последней секунды. Неприемлемо начать транзакцию из окна клиентской программы, а затем заставлять транзакцию ждать, пока она не получит некоторой дополнительной информации от пользователя прежде, чем сможет фактически завершить работу.

Пользователь при предоставлении этой информации может выказать чрезмерную медлительность или решить отключить компьютер, хотя транзакция продолжает выполняться. *Перерыв в транзакции* в конце концов приведет к откату транзакции, но это успеет негативно отразиться на общей производительности системы.

9.3.1.3. Автоматическое восстановление

Закон Мерфи гласит, что если что-то может испортиться, то оно испортится. Программы могут содержать ошибки, выполняющийся процесс может зависнуть или быть прекращен, электропитание может дать сбой, головки диска могут поломаться и т.д. К счастью, в большинстве ситуаций СУБД имеет возможность обеспечить *автоматическое восстановление* системы после отказа. Только в случае физической утраты данных

на диске требуется вмешательство АБД, чтобы дать указание СУБД о восстановлении данных с помощью последней *резервной копии* базы данных.

В зависимости от состояния транзакции в момент отказа СУБД автоматически выполняет *откат* (*rollback*) или *повтор всех завершенных операций* (*roll forward*) транзакции сразу после устранения причин возникшей проблемы. Восстановление осуществляется автоматически, однако, АБД может контролировать время, необходимое на восстановление, за счет установления частоты контрольных точек. *Контрольная точка* (*checkpoint*) — это процедура, которая заставляет СУБД временно приостановить все транзакции и записать все изменения, произведенные транзакциями (с момента запуска предыдущей контрольной точки) над базой данных.

На рис. 9.11 показаны основные моменты, связанные с автоматическим восстановлением системы после отказа [43]. Транзакция `t1` подтверждена после прохождения контрольной точки, но до отказа системы. Поскольку СУБД не известно, были ли физически записаны все изменения после контрольной точки в базу данных, она *повторяет все завершенные операции* транзакции `t1` после восстановления системы.

К транзакции `t2` применяется откат между контрольной точкой и отказом. Как и в случае транзакции `t1`, СУБД не известно, достигли ли диска изменения, связанные с откатом, — СУБД вновь выполняет *откат*.

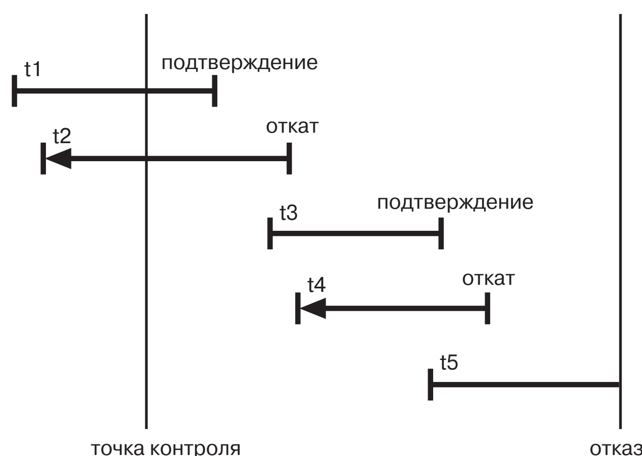

Рис. 9.11. Автоматическое восстановление

Остальные транзакции стартуют после контрольной точки. Для транзакции `t3` вновь повторяются все завершенные операции, чтобы убедиться, что произведенные ею изменения достигли базы данных. Аналогично транзакция `t4` повторяется, т.е. для нее выполняется *откат*.

Транзакция `t5` может не требовать никаких корректирующих действий со стороны СУБД, поскольку она выполнялась во время отказа. Любые изменения, произведенные транзакцией `t5` до отказа, не были записаны в базу данных. Все промежуточные изменения были записаны только в файл журнала. Пользователь поставлен в известность о том, что транзакция выполнялась во время отказа, и может перезапустить транзакцию после того, как СУБД восстановится и вновь начнет функционировать.

9.3.1.4. Программируемое восстановление

Хотя после непредвиденных отказов системы СУБД выполняет автоматическое восстановление, проектировщики и программисты должны контролировать и прогнозировать проблемы, связанные с транзакциями. СУБД обеспечивает набор возможностей по откату, которые можно задействовать программным способом, так что проблемы восстановления могут быть решены довольно изящно, при этом пользователь, возможно, даже не заметит, что в некоторый момент дела пошли не так.

Для начала заметим, что принципы разработки GUI-интерфейса, такие как контроль на стороне пользователя или терпимость к ошибкам (разд.7.3), требуют, чтобы программа позволяла пользователям совершать ошибки и была способна обеспечить восстановление системы. Откат, контролируемый программистом и применяемый в нужном месте программы, может восстановить предыдущее состояние базы данных (т.е. *отменить* ошибочные действия) при условии, что транзакция не подтверждена.

Если транзакция подтверждена, у программиста все еще остается возможность написать *компенсирующую транзакцию* (*compensating transaction*). Впоследствии пользователь может потребовать выполнения компенсирующей транзакции для отмены изменений в базе данных. Компенсирующие транзакции проектируются специально для того, чтобы разрешить программируемое восстановление, и должны быть отражены в моделях прецедентов.

9.3.1.4.1. Точка сохранения

Точка сохранения (*savepoint*) — это оператор программы, который делит длинную транзакцию на более короткие части. *Именованная точка сохранения* помещается в стратегически важных пунктах программы. Впоследствии программист имеет возможность осуществить откат выполненной работы к точке сохранения, а не к началу транзакции.

Например, программист может поместить точку сохранения прямо перед операцией `update`. Если операция `update` завершилась неуспешно, программа выполняет откат к точке сохранения и пытается повторить обновление. Вместо этого программа может предпринять любые другие действия, чтобы избежать полного аварийного прекращения транзакции.

В программах большого объема точки сохранения можно помещать перед каждой подпрограммой. Если подпрограмма отказывает, имеется возможность осуществить откат к началу подпрограммы и повторить ее выполнение с исправленными параметрами. При необходимости специально спроектированные и запрограммированные подпрограммы восстановления могут выполнить очистку, так что транзакция может возобновить выполнение.

9.3.1.4.2. Триггерный откат

Триггерный откат (*trigger rollback*) представляет собой особый вид точки сохранения. Как объяснялось в разд.9.1.2.2, триггер можно использовать для программирования бизнес-правил любой сложности. В некоторых случаях откат всей транзакции неприемлем, когда триггер отказывается модифицировать таблицу (из-за того, что транзакция пытается нарушить бизнес-правило). Транзакции может потребоваться предпринять корректирующие действия.

Вследствие приведенных выше причин СУБД может обеспечить возможности программирования триггера для отката либо всей транзакции, либо только триггера. В последнем случае программа (возможно, хранимая процедура) может проанализировать проблему и принять решение о дальнейших действиях. Даже если необходимо постепенно свернуть всю транзакцию, программа будет иметь возможность более тщательной интерпретации причины ошибки и отображения более осмысленного сообщения для пользователя.

9.3.1.5. Проектирование хранимых процедур и триггеров

Диаграмму навигации по программе (разд.9.2.2) можно расширить таким образом, чтобы включить в нее *состояния транзакции*. Это может быть довольно обременительно с точки зрения графической схемы. В качестве альтернативного решения можно разработать несколько диаграмм навигации по программе, каждая представляет определенный аспект навигации. Один из таких аспектов может быть посвящен фиксированию навигации в “среде” транзакций.

В любом случае диаграммы навигации по программе идентифицируют хранимые процедуры и триггеры. При этом необходимо отразить информацию, касающуюся цели, определения и детализированного проекта для каждой хранимой процедуры и триггера. В частности, для определения алгоритмов можно было бы применить нотацию некоторого псевдокода.

В качестве примера ниже приводится алгоритм для хранимой процедуры `DeleteEvent` приложения “Управление контактами с клиентами” (см рис. 9.10, разд.9.2.2). Процедура проверяет, является ли пользователь (работник), который пытается удалить информацию по мероприятию, также и создателем этой информации. Если это не так, операция удаления отклоняется. Процедура также проверяет, является ли мероприятие единственным оставшимся мероприятием для данного задания. Если это так, то задание также удаляется.

```
BEGIN
INPUT PARAMETERS (@event_id, @user_id)
Select Event (where event_id = @event_id)
IF @user_id = Event.created_emp_id
  THEN
       delete Event (where event_id = @event_id)
       IF no more events for
            Task.task_id = Event.task_id AND
            Event.event_id = @event_id
       THEN
            delete that Task
       ENDIF
  ELSE
       raise error ('Only the creator of the event can
                             delete that event')
ENDIF
END
```

Хранимая процедура `DeleteEvent` содержит операторы `delete` для удаления записей из таблиц `Event` и `Task`. Данные операторы `delete` должны запускать триггеры удаления на этих таблицах, если они существуют. Если алгоритм для этих триггеров выходит за пределы обычной проверки ссылочной целостности, проектировщик также должен предоставить для них спецификацию псевдокода (включая решение по стратегии отката — триггерный откат или откат транзакции).

9.3.2. Длинные транзакции

Некоторые новые классы приложений ИС поощряют кооперативные взаимодействия между пользователями. Эти приложения известны как приложения *вычислений для рабочих групп* (*workgroup computing*) или приложения *совместной работы, поддерживаемой компьютером* (*computer-supported cooperative work – CSCW*). В качестве примеров можно привести многие офисные приложения, системы творческой кооперации, системы автоматизированного проектирования, CASE-системы и т.д.

Во многих отношениях приложения для рабочих групп выдвигают требования к базам данных, которые являются ортогональными по отношению к традиционным моделям баз данных. Последние отличаются короткими транзакциями и изолируют пользователей друг от друга. Приложения для рабочих групп требуют длинных транзакций, управления версиями, управления кооперативной параллельностью и т.д.

Модель ОБД дает схему групповых вычислений, и многие продукты ОСУБД нацелены именно на эту область приложений. Пользователи, работающие с приложениями для рабочих групп, совместно используют информацию и осведомлены о работах, выполняемых ими над разделяемыми данными. Они работают в рамках собственной *рабочей среды*, используя персональные базы данных, включающие данные, *выбранные* (скопированные) из общей базы данных рабочей группы; работают с *длинными транзакциями*, которые могут занимать несколько сеансов работы с компьютером (пользователь может прерваться, затем возвратиться и продолжить работу с той же длинной транзакцией).

Отличительной чертой длинной транзакции является то, что для нее недопустим автоматический откат вследствие отказов без отслеживания системой хода транзакции. Чтобы понять это требование, представьте мое отчаяние, если бы для этого учебника сейчас был произведен "откат" из-за компьютерного сбоя! Откат для длинных транзакций контролируется пользователями с помощью процедур *точек сохранения*, которые постоянно сохраняют объекты в личной базе данных пользователя.

Понятие *короткой транзакции* не исчезло из сферы приложений для рабочих групп. Короткие транзакции необходимы для обеспечения атомарности и изолированности *во время* операций извлечения данных из групповой базы данных в личную базу и возврата данных (после их обработки в собственной рабочей среде) из личной базы данных в общую базу данных. После выполнения описанных коротких транзакций *короткие блокировки* освобождаются, а групповая база данных накладывает *длинные постоянные блокировки* на все извлеченные из нее объекты.

Модель длинных транзакций включает несколько взаимосвязанных целей, среди которых можно, в частности, выделить следующие [35], [52].

- Обеспечение обмена информацией (даже если она является временно противоречивой) между сотрудничающими пользователями.
- Обнаружение противоречий в данных и их согласованное разрешение.

- Использование преимуществ поддержки множества версий объектов для обеспечения управляемого совместного использования информации без потери результатов работы в случае отказа системы.

9.4. Замкнутое конструирование

Итеративный и наращиваемый процесс (разд.1.1.3.1) современного производства ПО требует четкой поддержки замкнутого конструирования, осуществляемого циклически между проектированием и реализацией. *Замкнутое конструирование* (*round-trip engineering*) определяется как сближение направленной вперед генерации программного кода и обратного конструирования, направленного от анализа программного кода к моделям проектирования. Замкнутое конструирование дает "...возможность работать как с графическим, так и с текстовым представлением, в то время как инструментальные средства поддерживают непротиворечивость обоих представлений" [8].

В случае приложений клиент-сервер замкнутое конструирование применяется по отдельности к *клиентским* прикладным программам и *серверным* программам баз данных. Возможно и, конечно, желательно применение различных CASE-средств для проектирования клиентской и серверной частей одного и того же проекта. Кроме того, CASE-средства, применяемые для замкнутого конструирования, должны быть тесно интегрированы с конкретной клиентской и серверной средами программирования.

9.4.1. Замкнутое конструирование для клиентских программ

Принципы замкнутого конструирования для клиентских приложений относительно просты — "дьявол кроется в деталях". На рис. 9.12 представлена диаграмма видов деятельности для типичного цикла замкнутого конструирования с использованием объектного языка программирования [61], [68]. UML-модели хранятся в CASE-репозитории (как мимоходом было объяснено, CASE-репозиторий представляет собой системы баз данных, часто — объектную базу данных).

`Проектная UML-модель` (UML-состояние) должна быть специально разработана с ориентацией на язык программирования. `Генерация кода` (вид деятельности UML) использует `Проектную UML-модель` для генерации `Исходного кода` (файлов заголовков и реализации). Любые поддерживаемые пользователем определения и вспомогательные объявления, внесенные на предыдущих итерациях замкнутого конструирования, подлежат сохранению.

`Исходный код` позже подвергается обычным для программы изменениям и расширениям. `Модифицированный код` подвергается процедуре реинжиниринга, результат которой представлен в виде `UML-модели реализации`. Вид деятельности, соответствующий процедуре реинжиниринга, на рис. 9.12 обозначен как `Генерация UML`. Для выявления изменений в проекте с помощью сравнения текущей `UML-модели реализации` и последней `Проектной UML-модели` используется некоторый вид инструментального средства под названием `Различение моделей`. Все принятые изменения распространяются на `Проектную UML-модель`, после чего возможно начинать следующую итерацию замкнутого конструирования.

Как мы заметили выше, "дьявол кроется в деталях". На практике многие связующие звенья между "пониманием" языка программирования CASE-средствами и реалиями конкретного компилятора для этого языка зачастую оказываются утерянными. CASE-

средство может сообщить о наличии ошибок в чисто скомпилированном коде, поскольку оно не распознает нюансов, характерных для конкретной версии компилятора.

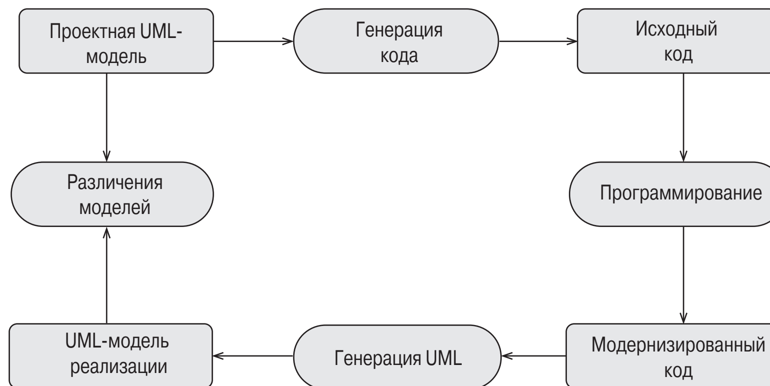

Рис. 9.12. Замкнутое конструирование в случае клиентской программы

CASE-средство может оказаться не в состоянии разрешить некоторые ссылки на объявления в зависимых от компилятора или библиотеки исходных файлах. Например, ссылки могут быть неразрешимы, если исходный файл ссылается на символы, определенные в другом файле и явно не включенные в данный файл. Хотя для подобных проблем и могут существовать обходные пути, они неизменно приводят к неидеальным решениям, требующим ручной корректировки.

Указанные трудности не означают, что от проведения замкнутого конструирования для клиентских программ следует вообще отказаться. Недоработки можно откорректировать вручную, и уже откорректированные UML-модели дают разработчикам надлежащую документацию для текущей версии программы. Альтернативой является полная потеря контроля над реализацией приложения.

9.4.2. Замкнутое конструирование для баз данных

В процессе замкнутого конструирования для баз данных [53] со стороны проекта участвует *физическая модель данных* (*ФМД*) (*physical data model— PDM*), а со стороны реализации — *база данных* (*БД*). Модель используется вместо UML-модели, поскольку UML не поддерживает физического проектирования баз данных (разд.8.4). Физическая модель данных должна быть ориентирована на конкретную СУБД.

На рис. 9.13 представлена диаграмма видов деятельности для замкнутого конструирования реляционной базы данных. После построения ФМД для базы данных (состояние `Начальная ФМД`) ее можно поместить в архив (вид деятельности `Архив`). Сравнение архивной и текущей ФМД позволяет зафиксировать последние изменения в модели.

Прямое конструирование модели `Начальная БД` на основе модели `Начальная ФМД` осуществляется с помощью вида деятельности `Создание БД`. В результате этой деятельности генерируется схема БД, включая триггеры. `Начальная БД` заполняется данными с помощью рекурсивного вида деятельности `Загрузка`. Вид деятельности `Модификация БД` вводит изменения в схему и вызывает переход базы данных в состояние `Текущая БД`.

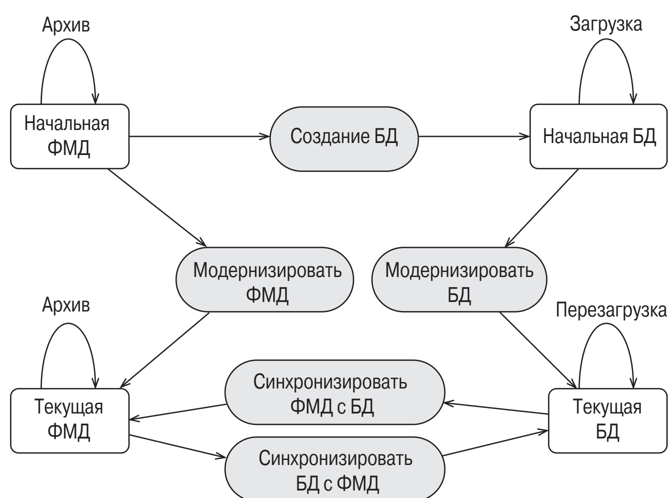

Рис. 9.13. Замкнутое конструирование в случае базы данных

Рис. 9.14. Экран сеанса реинжиниринга по преобразованию БД Sybase в ФМД PowerDesigner

На этом этапе существуют две возможности: изменения в `Текущей ФМД` могут быть синхронизированы с `Текущей БД` (вид деятельности `Синхронизировать ФМД с БД`) или

наоборот — изменения в `Текущей БД` могут быть синхронизированы с `Текущей ФМД` (вид деятельности `Синхронизировать БД с ФМД`). В результате этих видов деятельности может потребоваться перенести в архив `Текущую PDM`, а `Текущую БД` перегрузить.

На рис. 9.14 показан один из экранов сеанса реинжиниринга (вид деятельности `Синхронизировать с БД`) `Текущей БД` (Sybase), в результате которого мы получаем `Текущую ФМД` (PowerDesigner).

В процессе замкнутого конструирования баз данных необходимо учитывать приведенные ниже факторы [53].

- Архивирование и поддержка версий ФМД может осуществляться с использованием CASE-средств, однако, типичная РСУБД или ОРСУБД не обладает встроенными возможностями поддержки версий БД (отличных от простого создания новой БД).
- После генерации начальной схемы БД база загружается данными, и начинается программирование серверной программы. Программистам необходима возможность модифицировать (или требовать модификации) схемы БД по мере того, как модификации постепенно синхронизируются с текущей моделью ФМД. Вид деятельности `Синхронизировать ФМД с БД` должен полностью выполняться во время синхронизации, после чего должно осуществляться архивирование PDM-модели.
- Любые последующие изменения в архивной PDM-модели, которые требуют синхронизации с БД, должны быть внесены методом прямого конструирования в новый экземпляр БД (при отсутствии версий БД).

9.4.3. Реинжиниринг реляционных баз данных в объектно-реляционные

В главе 8 упоминалось о том, что следующее поколение технологий баз данных должно основываться на модели ОРБД. Поэтому следует ожидать возрастания потребности в переносе существующих (унаследованных) реляионных баз данных на объектно-ориентированные платформы. *Реинжиниринг* (*reengineering*) — это процесс, направленный на изучение и изменение унаследованной системы для реконструкции ее проекта и ее повторной реализации в новом виде.

Реинжиниринг и замкнутое конструирование разделяют технологии прямой и обратной разработки. Разница заключается в том, что замкнутое конструирование связано с эволюционным развитием *новой системы*. Реинжиниринг предшествует замкнутому конструированию и связан с получением начальной ФМД в результате изучения *унаследованного программного кода*.

Сегодня большинство проектов по реинжинирингу направлены на то, чтобы переориентировать устаревшие системы на основе языка COBOL на технологию реляционных баз данных. В будущем роль унаследованных систем перейдет к реляционным базам данных, которые на основе реинжиниринга будут превращаться в решения на основе ОРБД или ОБД. Однако, рынок чистых ОБД вряд ли будет расти так же быстро, как рынок ОРБД — отчасти из-за естественной инерции в переходе к совершенно новым технологиям, а отчасти из-за того, что ОСУБД обращены прежде всего к рыночной нише систем мультимедиа и кооперативной обработки.

На рис. 9.15 представлена диаграмма, отображающая последовательность видов деятельности в процессе реинжиниринга, целью которого является переход от сис-

темы РБД к системе ОРБД. Здесь также показан последовательный процесс замкнутого конструирования ОРБД.

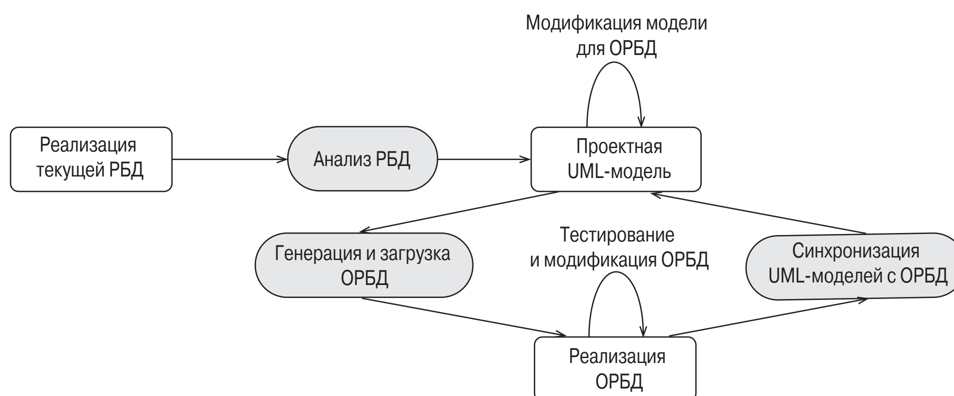

Рис. 9.15. Реинжиниринг реляционной базы данных в объектно-реляционную

Выполняется анализ `Реализации существующей РБД` (вид деятельности `Анализ РБД`) для определения структур базы данных, бизнес-правил и логики прикладных программ. Анализ приводит к построению проектной UML-модели, которая отражает текущее состояние существующей (унаследованной) реализации РБД. UML-модель затем модифицируется для `Реализации ОРБД`.

Теперь можно сгенерировать схему ОРБД, триггеры и методы (процедуры) и загрузить ОРБД данными. Для ОРБД создаются и тестируются прикладные программы. Это неизбежно приводит к изменениям в ОРБД, которые требуется отразить в проектной UML-модели. После этого можно сказать, что изменения подверглись реинжинирингу и первая итерация замкнутого конструирования завершена.

Процесс конструирования и замкнутого конструирования объектно-реляционных баз данных сталкивается с двумя основными проблемами. Первая касается аспектов замкнутого конструирования, а вторая связана с выразительными возможностями моделей UML.

Переход от проектнх моделей UML к реализациям ОРБД в процессе замкнутого конструирования аналогичен замкнутому проектированию для реляционной базы данных, как было рассмотрено в предыдущем разделе. Несмотря на аналогию, основополагающие технические вопросы для ОРБД значительно более трудны. Это связано с тем, что модель ОРБД более сложна и семантически более богата, чем модель РБД (глава 8). Начнем с того, что CASE-средства должны быть ориентированы на новый стандарт SQL 1999 и “понимали” различные варианты SQL 1999 применительно к коммерческим ОРСУБД.

Так же, как в случае модели РБД, рассчитывать на поддержку возможностей ФМД для систем ОРБД особенно не приходится. UML — это преимущественно язык анализа систем. Существуют профили UML, предназначенные для моделирования систем [56], однако, они не обеспечивают возможностей детализированного конструирования, связанных с пониманием всех нюансов ОРСУБД, включая нюансы, относящиеся к различным версиям одной и той же ОРСУБД.

Например, СУБД Oracle 8 использует объектные представления для облегчения перехода от реляционной к объектно-реляционной базе данных. Подобно другим объектам ОРБД, объектное представление может быть обозначено как класс-стереотип и ему, возможно, может быть присвоена собственная графическая пиктограмма. Безусловно стереотипному классу должна быть придана собственная точная семантика, так что CASE-средство может понять любые ограничения и возможности, связанные со стереотипным классом (например, какие типы отношений и других связей допустимы между стереотипным классом и любым другим класом модели). Низкоуровневые CASE-средства от компании Oracle (в данном случае) имеют значительно больше шансов помочь правильно решить подобные вопросы, чем исходный язык UML наряду со стереотипами.

Резюме

В этой главе мы остановились на некоторых более внутренних концепциях проектирования программ и транзакций и описали проектную деятельность, которая лежит на линии перехода между проектированием и реализацией. Были рассмотрены диаграммы навигации по программе. Мы также объяснили подход к замкнутому конструированию клиентских программ и серверных программ баз данных.

Программа, спроектированная надлежащим образом, отличается высоким уровнем *связности классов* при низком уровне *увязки классов*. Этого требования в отношении связности и увязки можно достичь, если проект подчиняется *закону Деметра*, который определяет допустимые целевые объекты для сообщений в рамках методов классов. Чрезмерное использование *методов открытия доступа* может привести к появлению "*бессмысленных классов*". Хотя *связность смешанных экземпляров* и представляется нежелательным явлением, иногда она все же допустима, поскольку среды программирования не поддерживают *динамическую классификацию*.

При разработке кооперации клиент-сервер необходимо принимать во внимание существование пяти уровней *SQL-интерфейсов*. Особый интерес представляет *SQL уровня 5*, поскольку позволяет пользователю непосредственно программировать базу данных. *Хранимые процедуры* и *триггеры* оказывают существенное влияние на серверный аспект проектирования программы. *Диаграммы навигации по программе* расширяют диаграммы навигации по окнам, позволяя учитывать наличие базы данных.

Транзакция — это единица работы базы данных, которая начинается, когда база данных находится в непротиворечивом состоянии, и по завершении которой база данных гарантированно окажется в непротиворечивом состоянии. Транзакции обеспечивают возможности по реализации параллелизма и восстанавливаемости баз данных. Традиционные приложения баз данных требуют коротких транзакций. Некоторые новые приложения баз данных работают с длинными транзакциями.

Замкнутое конструирование — это процесс, при котором проектные модели и программы синхронизированы, однако могут развиваться самостоятельно. Замкнутое конструирование определяется как сочетание направленной вперед генерации программного кода и обратного конструирования, направленного от анализа программного кода к моделям проектирования. Обычно замкнутое конструирование применяется по отдельности к *клиентским программам* и *программам баз данных*.

Вопросы

В1. Какое влияние на проектирование оказывают принципы, связанные со связностью и увязкой?

В2. Какие объекты могут выступать в качестве целевых объектов для сообщений согласно закону Деметра?

В3. Что такое “бессмысленный класс”?

В4. Объясните связь между динамической классификацией и связностью смешанных экземпляров.

В5. Кратко опишите пять уровней SQL-интерфейсов.

В6. В чем преимущество вызова из клиентской программы хранимой процедуры в сравнении с SQL-запросом, пересылаемым базе данных? Существуют ли ситуации, при которых мы вынуждены использовать SQL-запрос вместо вызова удаленной процедуры?

В7. Перечислите основные серверные объекты, моделируемые как состояния и виды деятельности диаграмм навигации по программе.

В8. Кратко опишите виды блокировок при пессимистическом управлении параллельностью.

В9. Кратко опишите уровни изолированности транзакций.

В10. Может ли проектировщик или АБД управлять продолжительностью восстановления базы данных? Объясните ваш ответ.

В11. Что такое компенсирующая транзакция? Как ее можно использовать при проектировании программы?

В12. Что такое точка сохранения? Как ее можно использовать при проектировании программы?

В13. В чем заключается функция дифференцирования модели при замкнутом конструировании клиентской программы?

В14. У традиционных баз данных отсутствуют встроенные возможности поддержки версий базы данных. Как это влияет на замкнутое конструирование программ для подобных баз данных?

Упражнения

У1. *Телемаркетинг* — обратитесь к разбору примера “Телемаркетинг”, как это определено в диаграмме видов деятельности для книги, и к решению примера У4 в главе 7.

Расширьте диаграмму навигации по окнам для приложения “Телемаркетинг” до диаграммы навигации по программе. Если решение оказывается слишком сложным, разделите диаграмму на несколько меньших диаграмм. Объясните, каким образом предложенное вами решение удовлетворяет требования к приложению.

У2. *Управление контактами с клиентами* — обратитесь к решениям примера “Управление контактами с клиентами”, как это определено в диаграмме видов деятельности для книги. В частности, рассмотрите диаграмму навигации по окнам в примере 7.5 (разд.7.6.2).

Идентифицируйте хранимые процедуры, триггеры и другие объекты базы данных (которые до сих пор не идентифицированы), необходимые для системы управления контактами с клиентами.