Глава 6

# Цикл существования объектов модели

Каждый объект имеет свой цикл существования. Объект “рождается”, затем, как правило, проходит ряд сменяющихся состояний, а потом “умирает” — либо удаляется, либо помещается в архив. Многие из объектов — простые и временные; они создаются после вызова конструктора, используются в вычислительных операциях, после чего их оставляют сборщику мусора. Усложнять такие объекты ни к чему. Но у некоторых объектов существование может продлиться намного дольше и частично пройти не в оперативной памяти. Они имеют сложные связи и отношения с другими объектами и проходят через изменения состояния, подчиняющиеся определенным инвариантам. Управление такими объектами не обходится без трудностей, которые легко могут привести к нарушению принципов ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

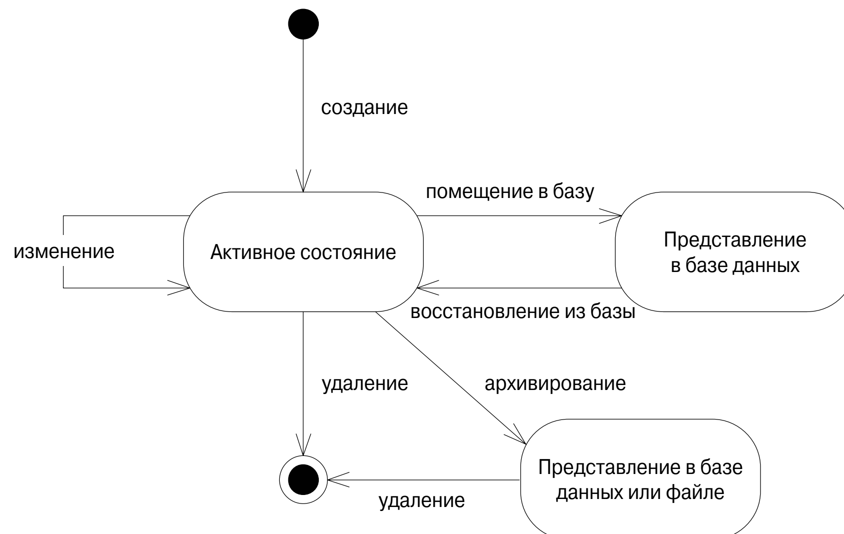


*Рис. 6.1. Цикл существования объекта из модели предметной области*

Эти трудности делятся на две категории.

1. Поддержание целостности объекта на этапе его существования.
2. Предотвращение излишней сложности в управлении циклом существования объектов.

В этой главе указанные проблемы будут решаться на основе трех архитектурных шаблонов. Во-первых, АГРЕГАТЫ (AGGREGATES) помогут “подтянуть” саму модель и избавиться от хаотической путаницы разнообразных объектов, четко определив права собственности и границы. Этот шаблон играет решающую роль в поддержании целостности объектов на всех этапах их существования.

Далее мы сместим акцент на начало цикла существования и воспользуемся ФАБРИКАМИ (FACTORIES) для создания и восстановления сложных объектов и АГРЕГАТОВ с сохранением инкапсуляции их внутренней структуры. Наконец, середина и конец жизненного цикла объектов — это компетенция ХРАНИЛИЩ (REPOSITORIES), которые позволяют находить и извлекать постоянно хранимые объекты, при этом инкапсулируя гигантскую инфраструктуру, стоящую за этими операциями.

Хотя ХРАНИЛИЩА и ФАБРИКИ непосредственно не происходят из предметной области, они играют важную смысловую роль в ее архитектуре. Эти конструкции — ценное подспорье в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ, поскольку они дают нам в руки удобные средства обращения с объектами модели.

Моделирование АГРЕГАТОВ наряду с добавлением в программу ФАБРИК и ХРАНИЛИЩ дает нам возможность манипулировать объектами систематически, в естественных смысловых границах, на протяжении всего цикла их существования. АГРЕГАТЫ обозначают область действия, в пределах которой на каждом этапе жизненного цикла должны удовлетворяться определенные инварианты. ФАБРИКИ и ХРАНИЛИЩА оперируют АГРЕГАТАМИ, инкапсулируя сложности специфических переходных этапов их существования.

## Агрегаты

Если снизить до минимума количество вводимых в модель ассоциаций, становится легче проследить взаимосвязи между понятиями и хотя бы немного ограничить их лавинообразный рост. Но большинство практических предметных областей обладает такой богатой внутренней связностью, что в конце концов все равно приходится трассировать

длинные цепочки ссылок одних объектов на другие. В каком-то смысле это отражение богатства реального мира, который нечасто балует нас четкими и нерушимыми границами. Но это-то и создает проблему при разработке программного обеспечения.

Предположим, мы удаляем объект **`Человек`** (**`Person`**) из базы данных. Вместе с ним удаляется имя, дата рождения, описание его работы. А как насчет адреса? По тому же адресу могут проживать и другие люди. Если удалить адрес, соответствующие объекты **`Человек`** будут содержать ссылки на удаленный объект. А если оставить, в базе будут накапливаться отработанные адреса. Автоматическая сборка мусора могла бы ликвидировать эти адреса, но даже если такая возможность есть в СУБД, это чисто техническая мера, — а принципиальный вопрос моделирования остается нерешенным.

Даже для отдельной транзакции трудно предсказать, до каких пределов может простираться ее влияние, учитывая переплетение взаимосвязей в типичной объектной модели. А обновлять каждый объект в системе на случай наличия какой-то связи — не слишком практично.

Проблема особенно обостряется в системах с параллельным доступом к одним и тем же объектам со стороны многочисленных клиентов. В случае, когда сразу много пользователей могут просматривать и изменять различные объекты в системе, приходится думать, как предотвратить одновременные модификации взаимозависимых объектов. Если неправильно оценить области определения и действия, можно оказаться в беде.

**При внесении изменений в объекты модели, обладающей сложной системой ассоциаций, трудно гарантировать согласованность этих изменений. Необходимо соблюдать инварианты, относящиеся к тесно связанным группам объектов, а не отдельным объектам. Но если применить слишком осторожные схемы блокирования, то параллельные пользователи станут невольно мешать друг другу, и это сделает всю систему неработоспособной.**

Сформулируем задачу по-другому. Откуда мы знаем, где начинается и где заканчивается объект, составленный из других объектов? В любой системе с постоянным хранением данных у каждой транзакции, которая вносит в данные изменения, должна быть своя ограниченная область действия. Кроме того, должен быть способ поддерживать согласованность данных (путем соблюдения их инвариантов). В базах данных можно применять разные схемы блокирования, а также программировать тесты. Но эти половинчатые решения отвлекают нас от модели, и очень скоро положение только усугубится.

На самом деле, чтобы найти разумное решение проблем такого рода, требуется углубленное понимание предметной области, а именно таких ее параметров, как частота смены экземпляров того или иного класса. Необходимо построить модель, которая давала бы больше свободы в местах интенсивной передачи конкурирующих данных, но заставляла четко соблюдать строгие инварианты.

Эта проблема, на первый взгляд, кажется технической, связанной с транзакциями баз данных, но корни ее лежат в модели — в недостаточно определенных границах. Если вывести решение из модели, она сама станет понятнее, и выразить ее в программной архитектуре станет легче. По мере пересмотра модели соответствующие изменения будут вноситься и в программную реализацию.

Существуют проработанные схемы для определения прав собственности/владения в модели. Описанная ниже простая, но строгая система, выведенная из них, предлагает набор правил для реализации транзакций, которые вносят изменения и в первичные объекты, и в те, которые ими владеют[1].

---

[1] Эту систему разработал и реализовал Дэвид Сигел (David Siegel) в проектах 90-х годов, но не опубликовал ее.

Прежде всего нам потребуется абстракция для инкапсуляции ссылок в пределах модели. Совокупность взаимосвязанных объектов, которые мы воспринимаем как единое целое с точки зрения изменения данных, называется АГРЕГАТОМ (AGGREGATE). У каждого АГРЕГАТА есть *корневой объект* и есть *граница*. Граница определяет, что находится внутри АГРЕГАТА. Корневой объект — это один конкретный объект-СУЩНОСТЬ (ENTITY), содержащийся в АГРЕГАТЕ. Корневой объект — единственный член АГРЕГАТА, на который могут ссылаться внешние объекты, в то время как объекты, заключенные внутри границы, могут ссылаться друг на друга как угодно. СУЩНОСТИ, отличные от корневого объекта, локально индивидуальны, но различаться они должны только в пределах АГРЕГАТА, поскольку никакие внешние объекты все равно не могут их видеть вне контекста корневой СУЩНОСТИ.

Пусть, например, в авторемонтной мастерской используется программа, в которой построена модель автомобиля. Автомобиль представляет собой сущность с глобальной идентичностью — его необходимо отличать от всех остальных машин в мире, даже очень похожих. Для этой цели подходит номерной знак, поскольку это индивидуальный идентификатор, присвоенный каждому автомобилю. Нам может понадобиться проследить историю использования шин в их четырех возможных положениях на колесах, узнать их пробег и степень износа. Чтобы точно знать, где какая шина, их тоже следует сделать СУЩНОСТЯМИ. Но очень маловероятно, что их индивидуальность будет нас интересовать вне контекста конкретного автомобиля. Если заменить шины и отослать старые на переработку, то либо наша программа вообще перестанет их отслеживать, либо они станут анонимными членами кучи списанных шин. Их пробег уже никого не будет волновать. Но более уместно к теме будет заметить, что даже если шины установлены на машине, никто не попытается ввести в систему запрос на поиск конкретной шины, чтобы потом посмотреть, на какой машине она стоит. Наоборот, в базу данных будет послан запрос на поиск машины, а потом запрос на временную ссылку, ведущую к шинам. Поэтому автомобиль является корневой СУЩНОСТЬЮ в АГРЕГАТЕ, граница которого охватывает также и шины. С другой стороны, двигатели имеют собственные, выбитые на них серийные номера, по которым их иногда разыскивают независимо от автомобилей. В некоторых приложениях двигатели могут быть корневыми объектами своих собственных АГРЕГАТОВ.

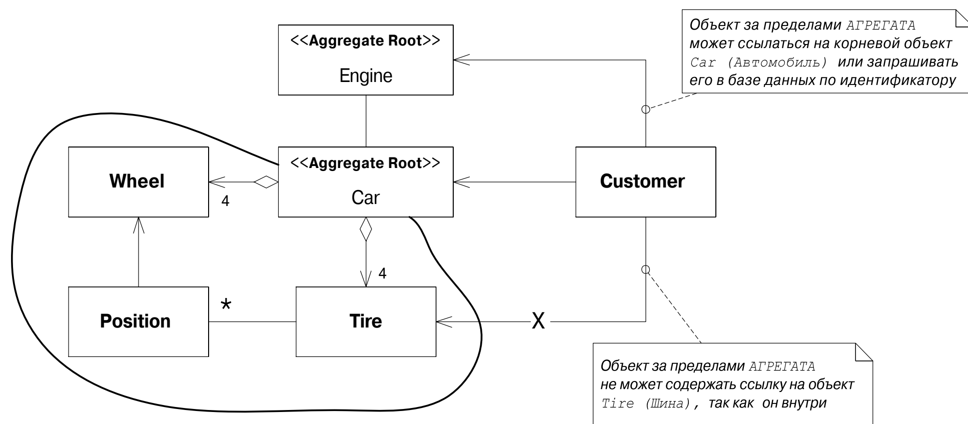


*Рис. 6.2. Локальная/глобальная идентичность и ссылки на объекты*

Из взаимосвязей между объектами АГРЕГАТА можно составить так называемые *инварианты*, т.е. правила совместности, которые должны соблюдаться при любых изменени-

ях данных. Не всякое правило, распространяющееся на АГРЕГАТ, обязано выполняться непрерывно. Восстановить нужные взаимосвязи за определенное время можно с помощью обработки событий, пакетной обработки и других механизмов обновления системы. Но соблюдение инвариантов, имеющих силу внутри агрегата, должно контролироваться немедленно по завершении любой транзакции.

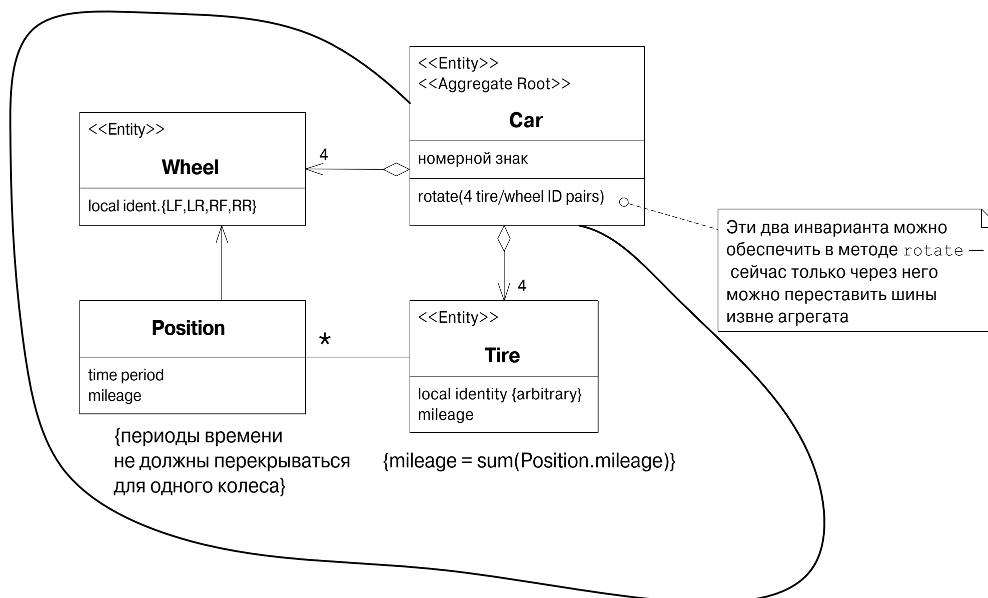


*Рис. 6.3. Инварианты* АГРЕГАТОВ

Чтобы реализовать концепцию АГРЕГАТА в виде практической программной конструкции, необходимо иметь набор правил, выполняющихся для любой транзакции.

- Корневой объект-СУЩНОСТЬ имеет глобальную идентичность и несет полную ответственность за проверку инвариантов.
- Некорневые объекты-СУЩНОСТИ имеют локальную идентичность — они уникальны только в границах АГРЕГАТА.
- Нигде за пределами агрегата не может постоянно храниться ссылка на что-либо внутри него, кроме его корневого объекта. Корневой объект-СУЩНОСТЬ может передавать ссылки на внутренние объекты-СУЩНОСТИ другим объектам. Но эти другие объекты могут использовать их только временно, и не имеют права хранить их или как-то фиксировать. Корневой объект может передать копию ОБЪЕКТА-ЗНАЧЕНИЯ другому объекту, и что с ней потом случится — не играет роли, поскольку это не более чем значение, и оно не имеет никаких связей с АГРЕГАТОМ.
- Как следствие из предыдущего правила, только корневые объекты АГРЕГАТОВ можно непосредственно получать по запросам из базы данных. Все остальные объекты разрешается извлекать только по цепочке связей.
- Объекты внутри АГРЕГАТА могут хранить ссылки на корневые объекты других АГРЕГАТОВ.

- Операция удаления должна одновременно ликвидировать все, что находится в границах АГРЕГАТА. (При наличии сборки мусора это просто. Поскольку внешних ссылок ни на что, кроме корневого объекта, не существует, достаточно удалить корневой объект, а остальное будет подчищено при сборке.)
- Как только вносится изменение в любой объект внутри границ АГРЕГАТА, следует сразу удовлетворить все инварианты этого АГРЕГАТА.

**Группируйте** СУЩНОСТИ **и** ОБЪЕКТЫ-ЗНАЧЕНИЯ **в** АГРЕГАТЫ **и определяйте границы каждого из них. Выберите один объект-**СУЩНОСТЬ **и сделайте его корневым. Осуществляйте все обращения к объектам в границах** АГРЕГАТА **только через его корневой объект. Разрешайте внешним объектам хранить ссылки только на корневой объект. Ссылки на внутренние объекты** АГРЕГАТА **следует передавать только во временное пользование, на время одной операции. Поскольку доступ к объектам** АГРЕГАТА **контролируется через корневой объект, неожиданные изменения внутренних объектов невозможны. В такой схеме разумно требовать удовлетворения всех инвариантов для объектов в** АГРЕГАТЕ **и для всего** АГРЕГАТА **в целом при любом изменении состояния.**

Было бы очень удобно иметь такую среду программирования, которая бы позволяла объявлять АГРЕГАТЫ, автоматически реализуя схемы блокирования и т.п. Не имея такой поддержки, группа разработчиков должна иметь достаточно самодисциплины, чтобы выработать соглашение об АГРЕГАТАХ и программировать в четком соответствии с ним.

## Пример

### Целостность товарного заказа

Рассмотрим потенциальные осложнения, которые могут возникнуть в упрощенной системе приема товарных заказов.

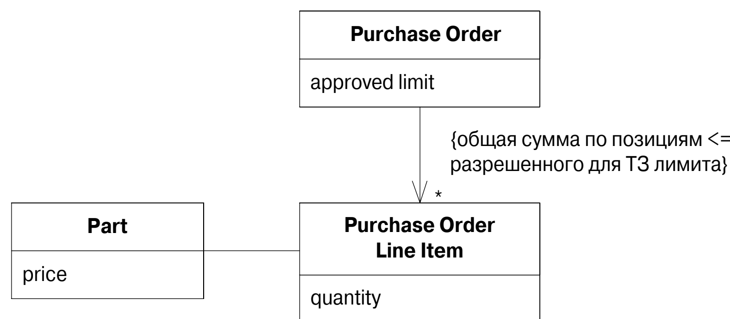


*Рис. 6.4. Модель системы по обработке товарных заказов*

На рисунке показана довольно обычная схема товарного заказа (ТЗ), разбитого на позиции, для которого введено правило-инвариант: суммарная стоимость всех позиций не может превышать предел, установленный для одного ТЗ в целом. В существующей реализации имеется три взаимосвязанные проблемы.

1. *Удовлетворение инварианта.* При добавлении новой позиции объект-заказ проверяет сумму и помечает себя как недопустимый, если превышен лимит. Как мы увидим далее, этой защитной меры недостаточно.

2. *Управление изменениями.* Если ТЗ перемещается в архив или удаляется, позиции уходят вместе с ним, но модель не дает никаких указаний, где нужно остановиться в следовании по цепочке взаимосвязей. Непонятно также, как именно на процесс влияет изменение цены того или иного товара в разное время.
3. *Параллельное обращение к базе данных.* Обращения разных пользователей к базе данных могут привести к конфликту данных.

Сразу несколько пользователей будут одновременно вводить и обновлять несколько ТЗ, и нам необходимо не дать им помешать друг другу. Начнем с очень простой схемы, в которой мы блокируем любой объект, который начал редактировать пользователь, пока он не закончит свою транзакцию. Таким образом, пока Джордж правит позицию 001, Аманда не имеет к ней доступа. Она может редактировать любую другую позицию в любом другом ТЗ (включая и другие позиции из ТЗ, над которым работает Джордж).

| **PO #0012946** | Approved Limit: $1,000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 3 | Guitars | @100.00 | 300.00 |
| 002 | 2 | Trombones | @200.004 | 00.00 |
| | | | | |
| | | | | |
| | | | Total :7 | 00.00 |

*Рис. 6.5. Начальное состояние объекта ТЗ, хранящегося в базе данных*

Объекты считываются из базы данных и помещаются в виде экземпляров в область памяти каждого пользователя. Там их можно просматривать и редактировать. Блокировка базы запрашивается только тогда, когда начинается редактирование. Итак, и Джордж, и Аманда могут работать параллельно, пока они "не трогают" товарные позиции друг друга. И все идет хорошо... пока Джордж и Аманда не приступают к работе над разными строками одного и того же товарного заказа.

Джордж добавляет гитары через свой интерфейс

| **PO #0012946** | Approved Limit: $1000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | **5** | Guitars | @100.00 | **500.00** |
| 002 | 2 | Trombones | @200.00 | 400.00 |
| | | | | |
| | | | | |
| | | | Total: | 900.00 |

Аманда добавляет тромбон через свой интерфейс

| PO #0012946 | Approved Limit: $1,000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 3 | Guitars | @100.00 | 300.00 |
| 002 | **3** | Trombones | @200.00 | 600.00 |
| | | | | |
| | | | | |
| | | | Total: | 900.00 |

*Рис. 6.6. Одновременное редактирование, разные транзакции*

Пользователям и их программам кажется, что все хорошо, потому что они игнорируют изменения, внесенные в другие части базы данных во время их транзакций — ни одна из заблокированных одним пользователем позиций не влияет на работу другого пользователя.

| PO #0012946 | Approved Limit: $1,000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 5 | Guitars | @100.00 | 500.00 |
| 002 | 3 | Trombones | @200.00 | 600.00 |
| | | | | |
| | | | | |
| | | | Total: | **1,100.00** |

*Рис. 6.7. В итоговом ТЗ нарушается инвариант — лимит общей стоимости*

После того как оба пользователя сохраняют свои изменения, в базу данных записывается ТЗ, в котором нарушается инвариант модели предметной области. Не соблюдено важное правило делового регламента. И никто даже не догадывается об этом.

Очевидно, блокирование одной позиции в заказе не является достаточной защитной мерой. Если же блокировать весь заказ сразу, этой проблемы, очевидно, не возникнет.

**Джордж редактирует заказ у себя**

| PO #0012946 | Approved Limit: $1000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 5 | Guitars | @100.00 | 500.00 |
| 002 | 2 | Trombones | @200.00 | 400.00 |
| | | | | |
| | | | | |
| | | | Total: | 900.00 |

Для Аманды ТЗ 0012946 заблокирован

Изменения Джорджа зафиксированы

**Аманда получает доступ: видны изменения Джорджа**

| PO #0012946 | Approved Limit: $1000.00 | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 5 | Guitars | @100.00 | 500.00 |
| 002 | 3 | Trombones | @200.00 | 600.00 |
| | | | | |
| | | | | |
| | **Limit exceeded →** | | Total: | 1,100.00 |

*Рис. 6.8. Соблюдение инварианта при блокировании целого ТЗ*

Программа не позволит сохранить результат, пока Аманда не устранит проблему — например, повысит лимит или уберет из заказа одну гитару. Итак, проблема предотвращена. Это решение может быть приемлемым, если работа в основном ведется одновременно над множеством разных заказов. Но если обычно вместе редактируются разные позиции одного большого ТЗ, такая блокировка создаст неудобства.

Но даже в случае множества мелких заказов есть и другие способы нарушить инвариант. Зайдем со стороны товара. Если кто-нибудь изменит цену на тромбоны, пока Аманда добавляет их в свой заказ, разве это не вызовет нарушения инварианта?

Попробуем заблокировать и товар в дополнение к целому ТЗ. И вот что получится, если Джордж, Аманда и Сэм работают над *разными* ТЗ.

Джордж редактирует ТЗ

Гитары и тромбоны заблокированы

| **PO #0012946** Approved Limit: $1,000.00 | | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | **2** | Guitars | @100.00 | **200.00** |
| 002 | 2 | Trombones | @200.00 | 400 .00 |
| | | | | |
| | | | | |
| | | | Total: | 600.00 |

Аманда добавляет тромбоны; должна ждать Джорджа

Скрипки заблокированы

| **PO #0012932** Approved Limit: $1,850.00 | | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | 3 | Violins | @400. 00 | 1,200.00 |
| **002** | **2** | **Trombones** | **@200.00** | **400.00** |
| | | | | |
| | | | | |
| | | | Total: | 1,600.00 |

Сэм добавляет тромбоны; должен ждать Джорджа

| **PO #0013003** Approved Limit: $15,000.00 | | | | |
|---|---|---|---|---|
| 001 | 1 | Piano | @1,000.00 | 1,000 .00 |
| **002** | **2** | **Trombones** | **@200.00** | **400.00** |
| | | | | |
| | | | | |
| | | | Total: | 1,400.00 |

*Рис. 6.9. Как блокировка-перестраховка мешает людям работать*

Неудобства накапливаются, потому что данные о музыкальных инструментах (товарах) конкурируют за очередность изменений и происходит вот что.

**Джордж добавляет скрипки; должен ждать Аманду (!)**

| **PO #0012946** Approved Limit: $1,000.00 | | | | |
|---|---|---|---|---|
| Item # | Quantity | Part | Price | Amount |
| 001 | **2** | Guitars | @100.00 | **200.**00 |
| 002 | 2 | Trombones | @200.00 | 400.00 |
| **003** | **1** | **Violins** | **@400.00** | **400.00** |
| | | | | |
| | | | Total: | 1,000.00 |

*Рис. 6.10. Затор*

Итак, всем троим придется подождать.

В этот момент можно начать усовершенствование модели, добавляя в нее следующие знания из предметной области.

**1.** Товары используются во многих ТЗ (высокая конфликтность данных).

**2.** Изменения в данные о товарах вносятся реже, чем в товарные заказы.

**3.** Изменения в ценах товаров не обязательно распространяются на уже заполненные ТЗ. Это зависит от момента времени, когда было внесено изменение, по отношению к состоянию ТЗ.

Пункт 3 особенно очевиден, когда дело касается архивных, уже выполненных ТЗ. В них, конечно, должны указываться цены на момент заполнения заказов, а не текущие.

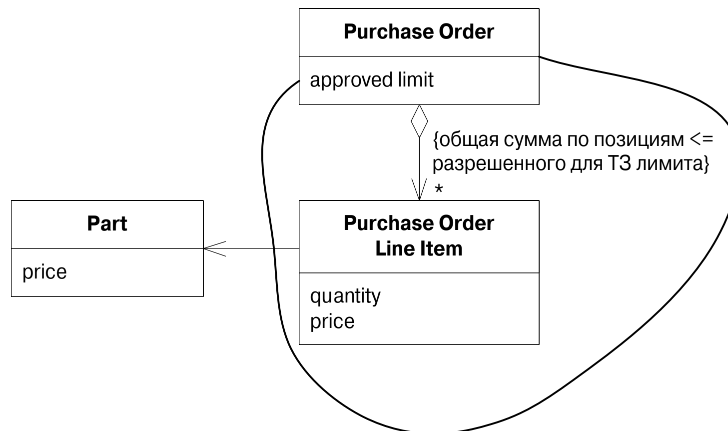


*Рис. 6.11. Цена копируется в* ***Позицию (Line Item)****. Теперь можно проконтролировать инвариант* АГРЕГАТА

Если привести программную реализацию в соответствие с этой моделью, то инвариант для ТЗ и его позиций будет выполняться гарантированно, а изменения в цене на товар не обязаны немедленно влиять на позиции, которые ссылаются на этот товар. Более широкие правила совместности можно учесть другими способами. Например, система могла бы каждый день предлагать пользователю список устаревших цен, чтобы их можно было обновить или удалить. Но это не инвариант, за выполнением которого нужен строгий контроль. Ослабляя взаимосвязь позиций в заказе и цен на товары, мы тем самым избегаем конфликтов данных и лучше отражаем реальные способы ведения дел. В то же время усиление взаимосвязи ТЗ и его позиций гарантирует выполнение важного делового регламента.

АГРЕГАТ вводит четкие права собственности на ТЗ и его позиции таким способом, который согласуется с реальной деловой практикой. Создание и удаление ТЗ и его позиций естественным образом объединены, тогда как создание и удаление товаров реализовано независимо.

* * *

АГРЕГАТЫ обозначают области действия, внутри которых на каждом этапе существования должны соблюдаться определенные инварианты. Описанные далее конструкции, а именно ФАБРИКИ (FACTORIES) и ХРАНИЛИЩА (REPOSITORIES), оперируют АГРЕГАТАМИ, инкапсулируя в себе некоторые сложные преобразования, выполняемые в жизненном цикле объектов.

## Фабрики

Если создание объекта или целого АГРЕГАТА представляет большую сложность или открывает постороннему глазу слишком много внутренней структуры, то нужную инкапсуляцию обеспечивают ФАБРИКИ (FACTORIES).

* * *

В значительной мере сила такого инструмента, как объекты, заключается в его сложном внутреннем устройстве, включая и ассоциации между его элементами. Объект следует "дистиллировать" (т.е. очищать от всего лишнего), пока в нем не останется ничего, что не имеет отношения к самой его сути и его роли в транзакциях. С объекта довольно и этих обязанностей, свойственных середине цикла его существования. Если на сложный объект возложить еще и ответственность за создание самого себя, то начинают возникать проблемы.

Двигатель автомобиля — весьма непростое техническое устройство, в котором десятки частей взаимодействуют друг с другом для выполнения главной задачи — вращения вала. Теоретически можно спроектировать такой двигатель, который сам бы брал поршни и вставлял их в цилиндры, а свечи зажигания в нем сами находили бы свои гнезда и ввинчивались в них. Но маловероятно, чтобы такая сложная машина была такой же надежной и работоспособной, как обычный двигатель. Вместо этого мы считаем нормальным, чтобы двигатель из частей собирал кто-то посторонний — это может быть автомеханик-человек или промышленный робот. Как робот, так и человек устроены значительно сложнее, чем двигатель, который они собирают. Работа по сборке из деталей не имеет ничего общего с работой по вращению вала. Сборщики работают только в процессе создания автомобиля — когда вы ведете его по дороге, вам не нужен ни механик, ни робот. Не бывает так, чтобы автомобиль собирали и вели одновременно, поэтому нет нужды в механизме, который бы объединял в себе обе эти функции. Аналогично, сборка сложного составного объекта — это такая работа, которую лучше отделить от обязанностей, которые объект будет выполнять впоследствии, в ходе своего существования.

Но перекладывание ответственности на другую заинтересованную сторону, объект-клиент, приводит к еще худшим последствиям. Клиент знает, какую работу нужно сде-

лать, и поручает необходимые вычислительные операции объектам предметной области. Если от клиента требуется самому собирать те объекты модели, которые для этого нужны, то он должен знать достаточно об их внутреннем устройстве. А чтобы соблюсти все инварианты, касающиеся взаимоотношений между отдельными частями объекта модели, клиент должен знать еще и некоторые внутренние правила (деловые регламенты) этого объекта. Даже простой вызов конструктора привязывает объект-клиент к конкретным классам объекта, который он создает. Никакие изменения в реализацию объектов модели теперь невозможно внести, не изменяя и объект-клиент, отчего становится труднее выполнять рефакторинг.

Когда создание объекта модели перекладывается на объект-клиент, это приводит к лишним сложностям и делает менее четким распределение обязанностей. К тому же, образуется брешь в инкапсуляции создаваемых объектов прикладной модели и АГРЕГАТОВ. Что еще хуже, если клиент находится на операционном уровне, то происходит "утечка" части обязанностей с уровня предметной области. Тесная привязка операционного уровня к подробностям реализации модели сводит на нет большинство преимуществ абстрагирования на уровне предметной области и сильно усложняет внесение дальнейших изменений.

**Создание объекта само по себе может быть очень важной операцией. Но сборка объекта — это совершенно не та работа, которую потом придется делать этому же объекту в ходе своего существования. Объединение этих обязанностей в одном объекте порождает неуклюжие, трудные для понимания программные конструкции. Если дать клиенту возможность управлять созданием объектов, это искажает архитектуру самого клиента, нарушает инкапсуляцию собранного объекта или** АГРЕГАТА**, а также слишком сильно привязывает клиента к конкретной реализации создаваемого им объекта.**

Создание сложных объектов — это обязанность уровня предметной области, но не объектов, которые непосредственно выражают модель. Бывают случаи, когда создание и сборка объекта соответствуют существенному событию в предметной области — например, "открытию банковского счета". Но сами по себе операции создания и сборки объекта обычно не имеют смысла в модели; это уже вопрос программной реализации. Чтобы решить эту проблему, приходится добавлять в модель конструкции, не являющиеся ни сущностями (ENTITIES), ни объектами-значениями (VALUE OBJECTS), ни службами (SERVICES). Здесь мы отходим от принципов предыдущей главы, поэтому важно подчеркнуть еще раз: мы добавляем в модель элементы, которые ничему в ней непосредственно не соответствуют, но, тем не менее, выполняют обязанности, относящиеся к уровню модели.

В любом объектно-ориентированном языке есть механизм создания объектов (например, конструкторы в Java и C++, класс создания экземпляров в Smalltalk). Но существует потребность и в более абстрактном механизме создания объектов, который не зависел бы от других объектов. Элемент программы, обязанность которого — создавать объекты, называется ФАБРИКОЙ (FACTORY).

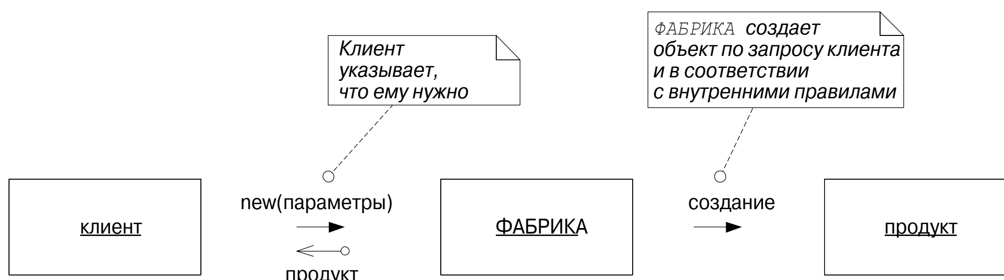


*Рис. 6.12. Работа с ФАБРИКОЙ*

Так же, как интерфейс объекта должен инкапсулировать его реализацию, т.е. давать возможность использовать операции объекта, не зная, как они устроены, ФАБРИКА инкапсулирует знания, необходимые для создания сложного объекта или АГРЕГАТА. Она предоставляет клиенту интерфейс, который соответствует его потребностям, и абстрактное представление созданного объекта.

**Передайте обязанности по созданию экземпляров сложных объектов и АГРЕГАТОВ отдельному объекту, который сам по себе может не выполнять никаких функций в модели предметной области, но, тем не менее, является элементом ее архитектуры. Обеспечьте интерфейс, который бы инкапсулировал все сложные операции сборки объекта и не требовал от клиента ссылаться на конкретные классы создаваемого объекта. Создавайте АГРЕГАТЫ как единое целое, контролируя выполнение инвариантов.**

* * *

ФАБРИКИ можно проектировать по-разному. В [14] подробно разобрано несколько специальных архитектурных шаблонов для создания объектов — FACTORY METHOD (МЕТОД-ФАБРИКА), ABSTRACT FACTORY (АБСТРАКТНАЯ ФАБРИКА) и BUILDER (КОМПОНОВЩИК). Изложение в упомянутой книге касается в основном самых трудных вопросов создания объектов. Наша же цель — не углубиться в подробности проектирования ФАБРИК, а показать, что они занимают важное место в архитектуре предметной области. Правильное пользование ФАБРИКАМИ — залог успешного ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

Любая хорошая фабрика должна отвечать двум фундаментальным требованиям.

1. Каждый метод создания объекта должен быть един и неделим; он должен гарантировать соблюдение всех инвариантов создаваемого объекта или АГРЕГАТА. ФАБРИКА должна уметь создавать только объект целиком в корректном состоянии. Для объекта-СУЩНОСТИ это означает создание сразу целого АГРЕГАТА с соблюдением всех инвариантов; только необязательные второстепенные элементы разрешается добавить позже. Для неизменяемого ОБЪЕКТА-ЗНАЧЕНИЯ это означает, что все атрибуты инициализируются окончательными корректными значениями. Если интерфейс позволяет клиенту запросить создание некорректного объекта, то должна инициироваться исключительная ситуация или запуститься какой-то другой механизм, не позволяющий возвратить некорректное значение.
2. Абстрагировать ФАБРИКУ следует к желаемому типу, а не к конкретному классу. В этом могут помочь оригинальные шаблоны фабрик, предлагаемые в [14].

## Выбор фабрик и их местонахождения

Грубо говоря, ФАБРИКА вводится для того, чтобы создавать некие объекты, скрывая подробности, и помещается туда, где ею будет удобно пользоваться. Принимаемые по этому поводу решения обычно касаются тех или иных АГРЕГАТОВ.

Например, если есть потребность добавлять элементы внутрь уже существующего АГРЕГАТА, можно создать МЕТОД-ФАБРИКУ в корневом объекте агрегата. Реализация внутренней структуры АГРЕГАТА таким образом скрывается от всех внешних клиентов, а поддержание целостности АГРЕГАТА при добавлении в него элементов поручается корневому объекту, как показано далее на рис. 6.13.

А вот еще один пример. МЕТОД-ФАБРИКУ можно поместить в объект, который непосредственно участвует в порождении другого объекта, хотя и не владеет им после создания. В том случае, когда при создании объекта преимущественно используются данные и, возможно, деловые регламенты другого объекта, такой подход позволяет сэкономить

на операции по извлечению информации из порождающего объекта с целью послать ее куда-то для создания объекта. Также при этом отражаются особые отношения между порождающим объектом и его продуктом.

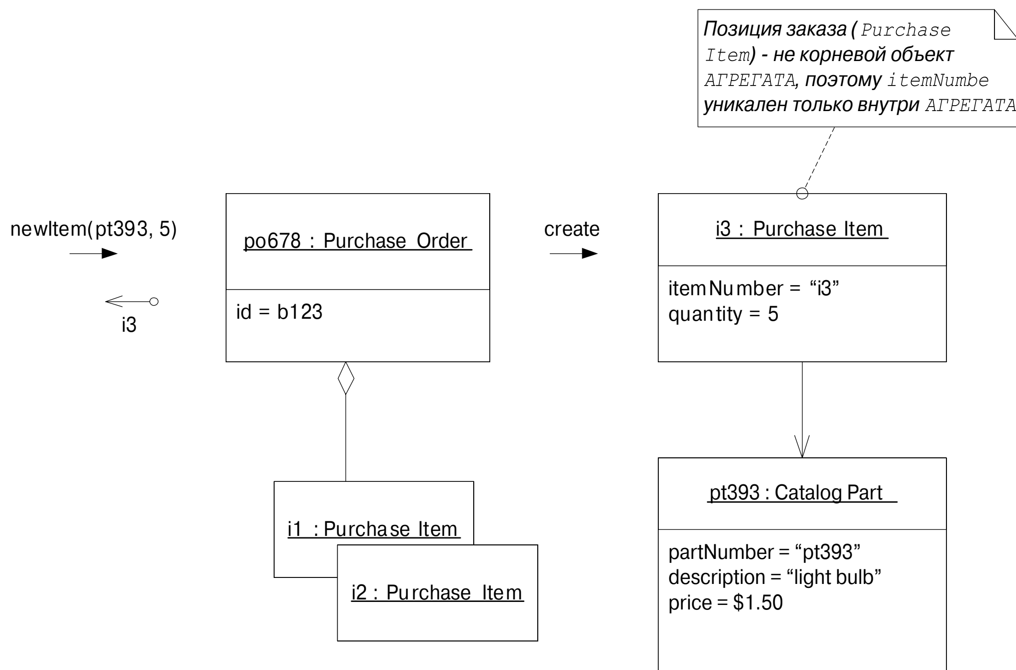


*Рис. 6.13. МЕТОД-ФАБРИКА инкапсулирует расширение АГРЕГАТА*

На рис. 6.14 объект **`Торговая заявка`** (**`Trade Order`**) не входит в тот же агрегат, что и **`Брокерский счет`** (**`Brokerage Account`**), потому что дальше он будет взаимодействовать с программой выполнения заявки, где **`Брокерский счет`** будет только мешать. Но даже с учетом этого кажется естественным дать **`Брокерскому счету`** право контроля над созданием **`Торговых заявок`**. **`Брокерский счет`** содержит информацию, которая должна помещаться в **`Торговую заявку`** (начиная с ее собственных идентификационных данных), а также правила, которые определяют, какие заявки являются правильными и разрешенными. Полезно было бы также скрыть реализацию **`Торговой заявки`**. Например, она может подвергнуться рефакторингу в иерархическую структуру, с отдельными подклассами для **`Заявки на продажу`** (**`Buy Order`**) и **`Заявки на покупку`** (**`Sell Order`**). Наличие же ФАБРИКИ избавляет клиента от привязки к конкретным классам.

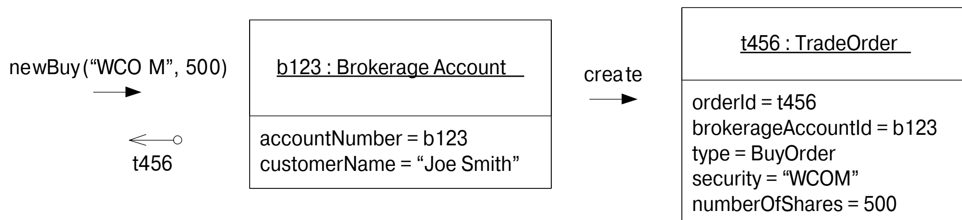


*Рис. 6.14. МЕТОД-ФАБРИКА порождает СУЩНОСТЬ, не входящую в тот же АГРЕГАТ*

ФАБРИКА очень тесно связана со своими "изделиями", поэтому и помещать ее нужно только в объект, который имеет сильную естественную связь с порождаемым объектом. Если нужно скрыть что-то — то ли конкретную программную реализацию, то ли просто сложность процесса создания объекта, — но не находится естественного объекта для инкапсуляции, нужно создать специальную ФАБРИКУ или СЛУЖБУ. Автономная ФАБРИКА обычно порождает сразу целый АГРЕГАТ, выдавая ссылку на его корневой объект и контролируя соблюдение инвариантов созданного АГРЕГАТА. Если нужна ФАБРИКА для объекта, внутреннего по отношению к некоторому агрегату, а корневой объект агрегата по каким-то причинам не подходит, смело создавайте для него отдельную ФАБРИКУ. Однако при этом надо уважать правила доступа внутри АГРЕГАТА и следить, чтобы на порождаемый объект были возможны только временные ссылки извне этого АГРЕГАТА.

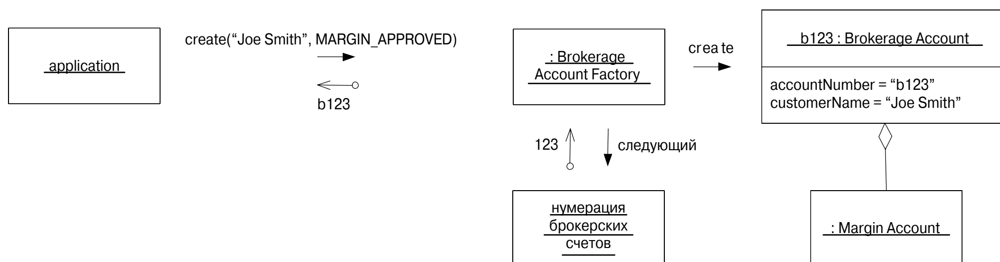


*Рис. 6.15. Построение АГРЕГАТА автономной ФАБРИКОЙ*

## Когда достаточно конструктора

Я видел чересчур много программ, в которых *все* экземпляры объектов создавались прямыми вызовами конструкторов классов или подобного же первичного механизма порождения объектов, имевшегося в языке программирования. Применение ФАБРИК имеет большие преимущества, но, в целом, используется реже, чем хотелось бы. И все же время от времени приходится делать выбор в пользу конструктора именно за его прямоту, потому что ФАБРИКИ могут усложнить работу с простыми объектами без полиморфизма.

Взвешивая "за" и "против", обычному общедоступному (*public*) конструктору нужно отдать предпочтение в следующих обстоятельствах.

- Класс является типом. Он не входит ни в какую интересную иерархию и не используется полиморфически для реализации интерфейса.
- Клиенту нужно знать реализацию объекта — возможно, с точки зрения выбора СТРАТЕГИИ.
- Все атрибуты объекта доступны клиенту, так что в конструкторе, предоставляемом клиенту, не создаются никакие новые объекты.
- Создание объекта не является сложным процессом.
- Общедоступный конструктор должен следовать тем же правилам, что и ФАБРИКА: это должна быть единая, неделимая операция, которая подходит для всех инвариантов создаваемого объекта.

Избегайте вызывать конструкторы в конструкторах других классов. Конструкторы должны быть проще простого. Сложную сборку, особенно АГРЕГАТОВ, поручите ФАБРИКАМ. И порог, за которым нужно вместо конструктора выбирать небольшой МЕТОД-ФАБРИКУ, совсем не высок.

В библиотеке классов Java имеются интересные примеры. Все коллекции реализуют интерфейсы, которые отделяют клиента от деталей реализации. И все же они создаются прямыми вызовами конструкторов. А ФАБРИКА могла бы инкапсулировать иерархию коллекций. Методы ФАБРИКИ могли бы позволить клиенту запрашивать нужные ему свойства и возможности, а ФАБРИКА выбирала бы, экземпляр какого класса создать. Тогда код для создания коллекций стал бы более выразительным, а новые классы-коллекции можно было бы устанавливать, не нанося удар по всем программам на Java.

Но есть и случай, говорящий в пользу конкретных конструкторов. Во-первых, выбор реализации может оказаться слишком затратным для многих программ, так что эти программы могут предпочесть делать его самостоятельно. (Хорошо построенная ФАБРИКА, тем не менее, могла бы и учесть такие факторы.) В конце концов, классов-коллекций существует не очень много, и выбор сделать не так уж трудно.

Абстрактные типы коллекций сохраняют некоторую ценность, несмотря на отсутствие ФАБРИК, по причине особенностей их использования. Очень часто коллекции создаются в одном месте, а используются в другом. Это означает, что клиент, который в конце концов пользуется коллекцией, — добавляет, удаляет и извлекает содержимое — имеет право общаться только с интерфейсом и не зависеть от реализации. Обязанность выбора класса коллекции обычно падает на объект, владеющий коллекцией, или на ФАБРИКУ владеющего объекта.

## Проектирование интерфейса

Разрабатывая декларативный заголовок ФАБРИКИ (неважно, автономной или ФАБРИКИ-МЕТОДА), не упускайте из виду два принципа.

- *Каждая операция должна быть единой и неделимой.* Все данные, которые необходимы для создания готового продукта-объекта, нужно передать за одну коммуникационную операцию с ФАБРИКОЙ. Придется также решить, что делать, если создание объекта сорвется, — например, в случае несоблюдения каких-то его инвариантов. Можно инициировать исключительную ситуацию или возвратить нулевое значение. Будьте последовательны и примите единый стандарт программирования для работы с ошибками создания объектов в ФАБРИКАХ.
- *Фабрика должна быть связана со своими аргументами.* Если неосторожно выбрать набор входных параметров, можно создать целую паутину взаимосвязей. На степень зависимости влияют операции, которые выполняются над аргументом. Если он просто вставляется в создаваемый объект, эта зависимость невелика. Но если при конструировании объекта из аргумента берутся фрагменты, зависимость становится сильнее.

Самые безопасные параметры — это те, которые поступают с нижних уровней архитектуры. Даже в пределах одного уровня существует тенденция разделения на подуровни: более элементарные объекты используются более сложными. (Это деление на уровни будет рассматриваться с разных сторон в главе 10, а затем снова в главе 16.)

Еще один хороший вариант параметра — это объект, который в модели является близкородственным генерируемому, так что между ними не создается новая взаимосвязь-зависимость. В предыдущем примере с объектом **`Позиция товарного заказа`** (**`Purchase Order Item`**) метод-фабрика принимает в качестве аргумента объект **`Товар из каталога`** (**`Catalog Part`**), который имеет естественную ассоциацию с **`Позицией`** (**`Item`**). Возникает прямая взаимосвязь между классом **`Товарный заказ`** (**`Purchase Order`**) и **`Товаром`** (**`Part`**). Но эти три объекта и так образуют замкнутую концептуальную группу. АГРЕГАТ **`Товарного заказа`** уже и так ссылался на **`Товар`**. Поэтому передача управления корневому объекту АГРЕГАТА и инкапсуляция внутренней структуры АГРЕГАТА — это хороший выбор в данном случае.

Используйте абстрактный тип аргументов, а не конкретные их классы. ФАБРИКА привязана к конкретному классу продуцируемых объектов; нет нужды привязывать ее еще и к конкретным параметрам.

## Где реализовать логику инвариантов?

Проследить, чтобы в создаваемом объекте или АГРЕГАТЕ выполнялись все инварианты — эта задача возложена на ФАБРИКУ. И все же нужно хорошенько подумать, прежде чем выносить свойственные объекту регламентные правила за его пределы. ФАБРИКА может делегировать проверку инвариантов самому своему продукту, и чаще всего лучше так и делать.

Но у фабрик устанавливаются особые отношения с создаваемыми ими объектами. Они уже знают внутреннюю структуру объектов, и сам смысл их существования в том и состоит, чтобы реализовать свой продукт. В некоторых обстоятельствах лучше как раз вынести логику инвариантов в ФАБРИКУ, не загромождая создаваемые объекты. Особенно удобно это делать с регламентными правилами АГРЕГАТОВ (которые распространяются на много объектов). А *неудобно* это делать с МЕТОДАМИ-ФАБРИКАМИ, которые включены в другие объекты модели.

Хотя в принципе инварианты проверяются в конце любой операции, иногда разрешенные над объектом преобразования просто не позволяют сделать такую проверку. Например, может существовать правило, регулирующее присвоение идентификационных данных объекту-СУЩНОСТИ. Но после создания объекта его данные могут оказаться неизменяемыми. Так, ОБЪЕКТЫ-ЗНАЧЕНИЯ целиком и полностью неизменяемы. Незачем объекту тащить на себе реализацию логики, которая никогда не будет применяться в течение его активного существования. В таких случаях инварианты лучше реализовать в ФАБРИКЕ, не усложняя ее продукт.

## Отличия между фабриками сущностей и фабриками объектов-значений

ФАБРИКИ СУЩНОСТЕЙ и ФАБРИКИ ОБЪЕКТОВ-ЗНАЧЕНИЙ отличаются друг от друга двумя особенностями. ОБЪЕКТЫ-ЗНАЧЕНИЯ неизменяемы; продукт создается в окончательном виде. Поэтому операции ФАБРИКИ должны давать полное описание продукта. ФАБРИКИ СУЩНОСТЕЙ же склонны работать только с самыми существенными атрибутами, необходимыми для создания корректного АГРЕГАТА. Детали можно добавить и позже, если они не требуются немедленно для соблюдения инварианта.

Есть еще проблемы, связанные с присвоением идентификационных данных СУЩНОСТИ — помимо ОБЪЕКТОВ-ЗНАЧЕНИЙ. Как говорилось в главе 5, идентификатор может назначаться программой автоматически или предоставляться снаружи — обычно пользователем. Если индивидуальность покупателя контролируется по номеру телефона, этот номер, очевидно, должен передаваться в виде аргумента в ФАБРИКУ. Соответственно, контроль над процессом присвоения идентификатора программой удобно возложить на ФАБРИКУ. Хотя фактическое генерирование уникального идентификационного номера обычно выполняется процедурой базы данных или другим инфраструктурным механизмом, ФАБРИКА знает, что именно запрашивать и куда это поместить.

## Восстановление хранимых объектов

В предыдущих разделах ФАБРИКА играла свою роль только в самом начале цикла существования объекта. В какой-то момент большинство объектов либо попадают в базу данных, либо передаются по сети. Притом очень немногие технологии баз данных сохраняют объектный характер своего содержимого — в большинстве способов передачи и

хранения данных объект приводится к более ограниченному представлению. Поэтому восстановление объекта в памяти — это сложный процесс сборки отдельных частей заново в единое целое.

ФАБРИКА, используемая для восстановления объекта, очень похожа на ФАБРИКУ для его создания, если не считать двух основных отличий.

1. *Фабрика, восстанавливающая объект-сущность, не присваивает ему новый идентификационный номер*. Если бы она это делала, то предыдущее воплощение объекта потерялось бы. Поэтому идентификационные атрибуты должны содержаться во входных параметрах ФАБРИКИ, занимающейся восстановлением объекта.
2. *Фабрика, восстанавливающая объект, по-другому обрабатывает нарушение инварианта*. В ходе создания нового объекта фабрика просто сбрасывает его, если не удовлетворяется инвариант, но при восстановлении требуется более гибкий подход. Если объект уже существует где-то в системе (например, в базе данных), этот факт нельзя просто проигнорировать. Но нельзя игнорировать и нарушение регламентов. Должна существовать какая-то схема для разрешения таких противоречий, отчего восстановление становится более сложной задачей, чем создание новых объектов.

На рис. 6.16 и 6.17 показаны два способа восстановления. Некоторые из требующихся для этого операций удобно реализованы в технологиях отображения объектов (*object mapping*), если речь идет о восстановлении из базы данных. Если же требуется более сложное восстановление из другого источника, то лучше воспользоваться специально предназначенной для этого ФАБРИКОЙ.

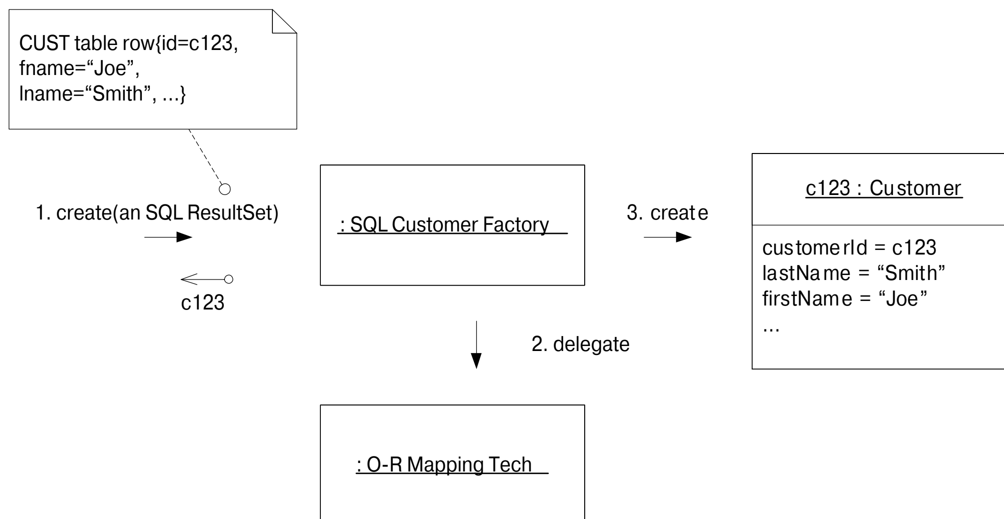


*Рис. 6.16. Восстановление объекта-СУЩНОСТИ, извлеченного из реляционной базы данных*

Подведем итоги. Для создания экземпляров объектов следует тщательно подобрать функциональные модули программы и явно определить их область действия. Это могут быть просто конструкторы, но часто возникает и потребность в более абстрактных или сложных механизмах создания экземпляров. Так в архитектуре программы появляется новый конструктивный элемент под названием ФАБРИКА (FACTORY). Обычно ФАБРИКИ явно не выражают никакую функциональную часть модели, но, тем не менее, входят в ее архитектуру как элементы, обеспечивающие четкую работу непосредственных смысловых объектов.

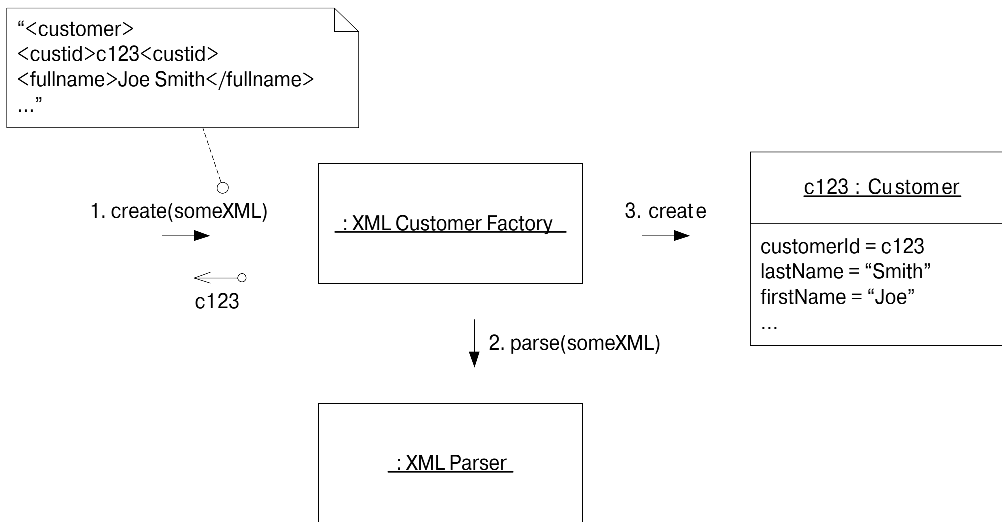


*Рис. 6.17. Восстановление объекта-СУЩНОСТИ, переданного в формате XML*

Фабрика инкапсулирует те преобразования в цикле существования объектов, которые связаны с их созданием и восстановлением. Но есть еще и преобразование, представляющее технические трудности и нетривиальное в архитектурной реализации — это сдача объектов на хранение и получение их оттуда. Это преобразование входит в обязанности еще одного конструктивного элемента модели — ХРАНИЛИЩА (REPOSITORY).

## Хранилища

Благодаря наличию ассоциаций можно найти один объект по его взаимосвязям с другими объектами. Но для этого нужно иметь отправную точку, отталкиваясь от которой можно проследить СУЩНОСТЬ или ОБЪЕКТ-ЗНАЧЕНИЕ в середине их цикла существования.

* * *

Чтобы делать с объектом что бы то ни было, нужно иметь ссылку на него. Как получить эту ссылку? Один из способов — создать объект, поскольку при создании объекта возвращается ссылка на него. Второй способ — проследить ассоциацию. Начинаем с объекта, который нам уже известен, и запрашиваем у него информацию о связанном с ним объекте. Во всякой объектно-ориентированной программе это происходит постоянно. Именно в таких взаимосвязях заключается основная выразительность объектных моделей. Но нужно где-то взять тот самый отправной объект, с которого все начинается.

Мне однажды попался проект, в котором разработчики, будучи энтузиастами ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ, пытались реализовать *любые* обращения к объектам через создание или прослеживание ассоциаций! Их объекты находились в объектной базе данных, и они рассудили, что существующие концептуальные взаимоотношения между ними автоматически обеспечат нужные ассоциации. Достаточно, дескать, только провести подробный анализ и сделать всю модель предметной области связной. Это добровольно наложенное на себя условие привело разработчиков к созданию как раз такого бесконечного переплетения связей, которого мы пытаемся избежать на протяжении вот уже нескольких глав, тщательно реализуя СУЩНОСТИ и подбирая АГРЕГАТЫ. Эта стратегия долго не продержалась, но разработчикам не удалось заменить ее каким-нибудь другим внятным и строгим подходом. В результате они нагромоздили сиюминутных решений и понизили планку своих амбиций.

Подобный подход мало кому пришел бы в голову, не говоря уже о том, чтобы взяться за его реализацию, поскольку в большинстве проектов информация хранится в реляционных базах данных. Такая технология хранения делает естественным третий способ получения ссылки на объект: для поиска объекта в базе выполняется запрос к ней по его атрибутам или же отыскиваются отдельные составляющие объекта, после чего он восстанавливается.

Поиск по базе данных глобально доступен и позволяет непосредственно перейти к любому объекту. Нет никакой нужды в том, чтобы все объекты были взаимосвязаны — достаточно урезать сеть взаимосвязей до приемлемого, управляемого состояния. Выбор между отслеживанием ассоциаций и поиском по базе становится рядовым проектным решением, в котором приходится балансировать между независимостью поиска и связностью ассоциаций. Должен ли объект **`Покупатель`** (**`Customer`**) хранить коллекцию всех своих **`Заказов`** (**`Orders`**)? Или же **`Заказы`** следует разыскивать в базе данных, используя поиск по полю **`Идентификатор покупателя`** (**`Customer ID`**)? Проектируя объекты, нужно выбирать разумное сочетание поиска и отслеживания ассоциаций.

К сожалению, обычно у разработчиков не доходят руки до таких тонкостей проектирования — им хотя бы разобраться в том море механизмов, которое необходимо для того, чтобы провернуть трюк с сохранением объекта и извлечением его обратно, а затем и удалением из места хранения.

С технической точки зрения извлечение хранимого объекта — это частный случай операции его создания, поскольку данные из базы используются фактически для сборки нового объекта. Действительно, код, который приходится для этого писать, не дает забыть об этой суровой реальности. Но с концептуальной точки зрения это всего лишь *середина* цикла существования объекта-СУЩНОСТИ. Ведь объект **`Покупатель`** (**`Customer`**) не стал представлять нового покупателя только потому, что объект положили в базу данных, а затем достали из нее. Чтобы не забывать об этом различии, мы и говорим о создании нового экземпляра на основе сохраненных данных как о *восстановлении* объекта.

Цель DDD состоит в том, чтобы научиться писать более качественные программы, сосредоточившись на модели предметной области, а не на программных технологиях. Пока

разработчик построит запрос SQL, передаст его в службу обработки запросов на инфраструктурном уровне, получит набор строк из таблицы базы данных, извлечет из них нужную информацию и передаст ее в конструктор или ФАБРИКУ, от его концентрации на модели ничего не останется. Так вырабатывается способ мышления об объектах всего лишь как о контейнерах данных, извлекаемых по запросам, и вся архитектура программы начинает ориентироваться на задачи обработки данных. Технические детали могут различаться, но главная проблема остается: клиентская часть взаимодействует с технологиями, а не с понятиями модели. Здесь могут оказаться полезными такие инфраструктуры, как УРОВНИ ОТОБРАЖЕНИЯ МЕТАДАННЫХ (METADATA MAPPING LAYERS) [13]. С их помощью облегчается преобразование результатов запросов в объекты, но и в этом случае программист попрежнему думает о технических механизмах, а не о предметной области. Что еще хуже, если клиентский код напрямую обращается к базе данных, у программистов возникает искушение вообще выбросить такие конструкции модели, как АГРЕГАТЫ, или даже инкапсуляцию объектов, и заняться прямой обработкой извлекаемых данных. Регламентные правила предметной области все больше перетекают в код запросов к базе данных, а то и попросту отбрасываются. Объектные базы данных, конечно, снимают проблему преобразования, но механизмы поискато все равно носят механистический характер, и разработчиков продолжают соблазнять перспектива "вытаскивать" из базы любые объекты по своему усмотрению.

**Клиентское приложение нуждается в удобных средствах получения ссылок на существующие объекты предметной области. Если инфраструктура позволяет это делать относительно легко, разработчики клиента добавляют в модель больше прослеживаемых ассоциаций, загромождая ее. С другой стороны, они могут с помощью запросов извлекать из базы данных в точности ту информацию, которая им нужна — например, достать несколько конкретных подобъектов, не обращаясь к корневому объекту** АГРЕГАТА**. Таким образом операционная логика предметной области переносится в запросы и клиентский код, а роль** СУЩНОСТЕЙ **и** ОБЪЕКТОВ-ЗНАЧЕНИЙ **сводится к простым контейнерам данных. Техническая сложность реализации всей инфраструктуры доступа к базе данных быстро загромождает код клиента, вынуждая разработчиков урезать и упрощать уровень предметной области. В итоге модель становится бесполезной.**

Если придерживаться изученных нами до сих пор принципов проектирования, можно попробовать несколько сузить рамки проблемы доступа к объектам. Вот бы иметь такой метод доступа, который позволил бы не отступить от этих принципов и сохранить именно модель в центре внимания при разработке программы. Для начала не следует беспокоиться о временных объектах. Такие объекты (обычно ОБЪЕКТЫ-ЗНАЧЕНИЯ) проживают короткую жизнь — они используются в операциях создавших их клиентов, а затем сбрасываются. Что касается постоянных объектов, то для работы с ними не надо использовать запросы к базе данных, если их удобнее находить по цепочке ассоциаций. Например, адрес человека можно узнать из объекта **`Человек`** (**`Person`**). И что самое важное, *обращение к любому объекту, внутреннему по отношению к* АГРЕГАТУ, *может выполняться только через корневой объект агрегата*.

Постоянные ОБЪЕКТЫ-ЗНАЧЕНИЯ обычно находятся путем отслеживания ассоциаций от какой-нибудь сущности, служащей корневым объектом для инкапсулирующего их АГРЕГАТА. Фактически доступ к ЗНАЧЕНИЮ через глобальный поиск часто не имеет смысла, поскольку нахождение ОБЪЕКТА-ЗНАЧЕНИЯ по его свойствам эквивалентно созданию нового экземпляра с теми же свойствами.

Впрочем, бывают и исключения. Например, если я планирую поездку с помощью транспортной онлайн-системы, то иногда сохраняю несколько потенциальных маршрутов, а позже возвращаюсь и выбираю один из них для резервирования билетов. Эти маршруты представляют собой ЗНАЧЕНИЯ (если бы существовало два маршрута, состоящих

из одних и тех же рейсов, то разницы между ними не было бы), но они ассоциированы с моим именем, и поэтому извлекаются в том же виде, в котором сформированы. Еще один случай — это перечислимые типы, область значений которых состоит из ограниченного количества заранее определенных величин или символов.

Надо сказать, что глобальный доступ к ОБЪЕКТАМ-ЗНАЧЕНИЯМ распространен значительно меньше, чем к сущностям. Так что если у вас возникает необходимость искать в базе данных существующий ОБЪЕКТ-ЗНАЧЕНИЕ, стоит подумать над тем, не представляет ли он на самом деле СУЩНОСТЬ, индивидуальность которой вы еще просто не осознали.

Из вышеизложенного должно быть ясно, что большинство объектов не требует глобального поиска для обращения к себе. А те, которые требуют, нужно соответствующим образом представить в архитектуре программы.

Теперь проблему можно сформулировать более точно.

**Необходимо, чтобы какая-то часть постоянно существующих объектов была доступна через глобальный поиск по атрибутам. Доступ таким методом осуществляется к корневым объектам агрегатов, которые неудобно отслеживать по ассоциациям. Обычно это** СУЩНОСТИ**, иногда —** ОБЪЕКТЫ-ЗНАЧЕНИЯ **со сложной внутренней структурой, а иногда значения из перечислимых типов. Предоставление такого доступа к другим объектам стирает основные различия между разновидностями объектов. Отсутствие ограничений на запросы к базе данных может нарушить инкапсуляцию объектов и** АГРЕГАТОВ **предметной области. Открытое использование технической инфраструктуры и механизмов доступа к базам данных усложняет работу клиентского приложения и знаменует отход от принципов** ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ**.**

Имеется множество приемов для решения технических проблем доступа к базам данных. Можно, например, инкапсулировать код SQL в ОБЪЕКТЫ-ЗАПРОСЫ (QUERY OBJECTS) или выполнять перевод из объектов в таблицы и назад через УРОВНИ ОТОБРАЖЕНИЯ МЕТАДАННЫХ [13]. Восстановление хранящихся объектов можно выполнять через ФАБРИКИ (об этом еще будет говориться). Эти и многие другие приемы дают возможность абстрагироваться от технических сложностей и тонкостей.

А теперь следует еще раз напомнить, что мы потеряли. Мы уже не думаем о понятиях нашей модели предметной области! Код уже не передает предметную суть выполняемых операций, а занимается пересылкой и обработкой данных. Шаблон ХРАНИЛИЩА (REPOSITORY) — это концептуально несложная архитектура, позволяющая инкапсулировать описанные выше технические решения и сконцентрировать внимание на прикладной модели.

ХРАНИЛИЩЕ представляет все объекты определенного типа в виде концептуального множества (обычно виртуального, эмулируемого). Оно работает аналогично коллекции, только с более развитым механизмом запросов. Можно добавлять и удалять объекты соответствующего типа, и скрытые механизмы ХРАНИЛИЩА будут автоматически помещать их в базу данных или удалять из нее. Введя это определение, получаем полный набор средств для доступа к корневым объектам АГРЕГАТОВ с самого начала и до конца их цикла существования.

Клиенты запрашивают объекты из ХРАНИЛИЩА, используя *запросные методы* (*query methods*), которые отбирают объекты по критериям, задаваемым клиентами, — обычно по значениям определенных атрибутов. ХРАНИЛИЩЕ выдает запрашиваемый объект, инкапсулируя механизмы запросов к базе данных и отображения метаданных. ХРАНИЛИЩА могут реализовать множество самых разных запросов, отбирая объекты в зависимости от установленных клиентами критериев. Они также могут возвращать информацию сводного характера, — например, сколько экземпляров объектов удовлетворяют тому или иному критерию. ХРАНИЛИЩЕ может даже выполнять вычисления по итогам запроса,

например, общую сумму или среднее каких-нибудь числовых атрибутов для объектов, полученных по этому запросу.

*Рис. 6.18. ХРАНИЛИЩЕ, выполняющее поиск по запросу клиента*

Наличие ХРАНИЛИЩА снимает с клиента огромную тяжесть — теперь он может общаться с простым, скрывающим технические подробности интерфейсом, и запрашивать нужную ему информацию в терминах модели. Для поддержки всего этого нужна развитая и сложная техническая инфраструктура, но сам интерфейс остается простым и концептуально привязанным к модели предметной области.

**Для каждого типа объектов, к которым требуется глобальный доступ, введите объект-посредник, который может создать иллюзию, что все объекты такого типа объединены в коллекцию и находятся в оперативной памяти. Наладьте доступ через хорошо известный глобальный интерфейс. Реализуйте методы для добавления и удаления объектов, инкапсулирующие реальное помещение информации в базу данных и удаление ее оттуда. Реализуйте методы, которые будут выбирать объекты по заданным критериям и возвращать полностью сгенерированные и инициализированные объекты или коллекции объектов с атрибутами, подходящими под критерии, таким образом инкапсулируя реальные технологии хранения данных и выполнения запросов. Реализуйте** ХРАНИЛИЩА **только для тех** АГРЕГАТОВ**, к корневым объектам которых требуется прямой доступ. Программа-клиент должна опираться на модель, а все операции хранения и обработки данных объектов должны быть переданы** ХРАНИЛИЩАМ**.**

* * *

У ХРАНИЛИЩ есть ряд важных преимуществ, ведь они:

- предоставляют клиентам простую модель для получения устойчиво существующих объектов и управления их жизненным циклом;
- убирают из операционной части приложения и модели предметной области необходимость в технической поддержке целостности объектов, разных вариантов технологий СУБД, и даже разных источников данных;
- выражают в себе проектные решения по способам доступа к объектам;
- позволяют легко заменить себя “заглушками” для целей тестирования (обычно заглушкой служит находящаяся в оперативной памяти коллекция).

## Запросы к хранилищам

Все ХРАНИЛИЩА должны содержать методы, с помощью которых клиенты могут запрашивать объекты, соответствующие некоторым критериям. Но вот в организации такого интерфейса возможны варианты.

Самое простое в разработке ХРАНИЛИЩЕ содержит явно прописанные в коде запросы с конкретными параметрами. Предназначение запросов может быть самым разнообразным: извлечение СУЩНОСТИ по ее идентификационным данным (это реализовано практически в любом ХРАНИЛИЩЕ); получение коллекции объектов по значению какого-либо атрибута или сложной комбинации параметров; отбор объектов по диапазону значений атрибутов (например, по датам); и даже различные расчеты, не выходящие за пределы общей компетенции ХРАНИЛИЩ (в виде интерфейса к операциям используемой СУБД).

Большинство таких запросов возвращает объект или коллекцию объектов, но в рамки концепции вполне вписывается и возврат результатов различных сводно-статистических вычислений, — например, количества объектов или суммы значений тех числовых атрибутов, которые в модели задействованы именно в такой операции.

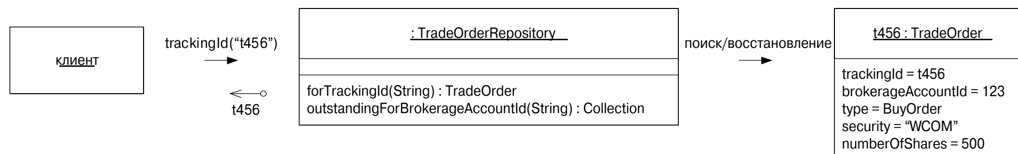


*Рис. 6.19. Явно прописанные запросы в простейшем ХРАНИЛИЩЕ*

Запросы, прописываемые явно, можно строить для любой инфраструктуры без особых затрат, потому что они делают всего-навсего то, что клиент все равно хочет от системы.

А вот в проектах с большим количеством запросов можно попытаться построить не просто ХРАНИЛИЩЕ, а целую архитектурную среду, ассоциированную с ним, в которой можно было бы составлять более гибкие запросы. Для этого потребуются кадры, знакомые с нужными технологиями, а также соответствующая техническая инфраструктура.

Одно из особенно удачных решений для обобщения ХРАНИЛИЩ путем создания архитектурной среды состоит в том, чтобы строить запросы на основе СПЕЦИФИКАЦИЙ (SPECIFICATION). СПЕЦИФИКАЦИЯ позволяет клиенту описывать (т.е. задавать, *специфицировать*), что именно ему нужно, и при этом не беспокоиться, как именно это делается. В процессе этого создается объект, который фактически и осуществляет нужный выбор. Этот архитектурный шаблон будет рассматриваться подробно в главе 9.

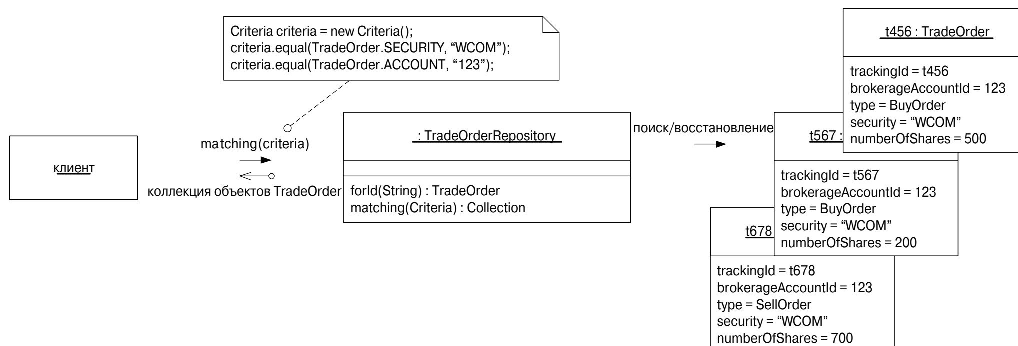


*Рис. 6.20. Гибкая декларативная СПЕЦИФИКАЦИЯ критериев поиска в сложном ХРАНИЛИЩЕ с расширенными возможностями*

Запросы на основе СПЕЦИФИКАЦИЙ составляются гибко и удобно. В зависимости от имеющейся инфраструктуры, архитектурная среда для хранилищ может быть или совсем простой, или непомерно усложненной. Проектирование таких хранилищ и связан-

ную с ним техническую проблематику подробнее рассматривают Роб Ми (Rob Mee) и Эдвард Хайет (Edward Hieatt) в [13].

Даже если хранилище спроектировано на выполнение гибких запросов, оно должно позволять также и добавлять специализированные, явно прописанные запросы. Это могут быть вспомогательные методы, инкапсулирующие часто используемые запросы, или такие, которые возвращают не сами объекты, а, например, результаты определенных математических операций с ними. Среды, которые не предусматривают такую возможность, в конце концов, либо "замутняют" чистоту архитектуры предметной области, либо просто игнорируются разработчиками.

## Клиентам безразлична реализация хранилищ, а разработчикам — нет

Реализация технологии длительного хранения и поддержания целостности объектов дает возможность приложению-клиенту быть очень простым и совершенно независимым от конкретной реализации ХРАНИЛИЩА. Но инкапсуляция инкапсуляцией, а часто бывает, что разработчикам необходимо знать, что происходит там внутри, "в глубине". ХРАНИЛИЩА могут работать и использоваться очень по-разному, и вопросы быстродействия часто играют ключевую роль.

Кайл Браун (Kyle Brown) рассказал мне историю о том, как к нему обратились для решения проблем с системой управления производством на основе WebSphere, которую он как раз устанавливали в ее рабочей среде. Система загадочно съедала всю память после нескольких часов работы. Кайл просмотрел код и нашел причину. В какой-то момент в системе собиралась сводная информация по всем учитываемым объектам на предприятии. Разработчики реализовали это в виде запроса под названием "все объекты", который создавал и инициализировал экземпляры всех объектов, а потом отбирал то, что было нужно. Получалось так, как будто вся база данных одним махом оказывалась в памяти! При тестировании этой проблемы не было, потому что объем тестовых данных был невелик.

Здесь недосмотр очевиден, но серьезные проблемы могут возникнуть и из-за гораздо более мелких недочетов. Разработчикам необходимо знать технические характеристики операций, инкапсулированных в объектах, — пусть и не обязательно мельчайшие подробности их реализации. Если компонент хорошо спроектирован, ему можно дать такую характеристику. (Это одна из главных тем в главе 10.)

Как говорилось в главе 5, на выбор тех или иных решений в моделировании могут повлиять особенности и ограничения инфраструктурной технологии. Например, наличие реляционной базы данных накладывает практические ограничения на глубокие сложносоставные объектные структуры. В общем, разработчики должны иметь достаточный доступ к потоку информации с обоих уровней: как по вопросам использования ХРАНИЛИЩА, так и по технической реализации его запросов.

## Реализация хранилища

Конкретные реализации ХРАНИЛИЩ могут сильно отличаться друг от друга в зависимости от имеющейся инфраструктуры и технологии контроля существования объектов. В идеале было бы хорошо спрятать от приложения-клиента (но не от разработчика этого клиента) все детали операций, чтобы код клиента оставался одним и тем же независимо от того, хранятся ли данные в объектной базе, реляционной базе или просто в оперативной памяти. ХРАНИЛИЩЕ же будет делегировать задания соответствующим службам инфраструктуры для выполнения порученной ему работы. Инкапсуляция механизмов

хранения данных, их извлечения и выполнения запросов — это самое основное в реализации ХРАНИЛИЩА.

Концепцию ХРАНИЛИЩА можно адаптировать ко многим ситуациям. Возможности ее реализации настолько разнообразны, что здесь будет приведено только несколько общих принципов, которые полезно помнить.

- *Абстрагируйте тип.* ХРАНИЛИЩЕ как бы "содержит в себе" все экземпляры определенного типа, но это не значит, что для каждого класса надо иметь свое хранилище. В качестве типа можно использовать абстрактный надкласс из иерархии. Например, **Товарная заявка** (**TradeOrder**) может быть как **Заявкой на покупку** (**BuyOrder**), так и **Заявкой на продажу** (**SellOrder**). Тип также может представлять собой интерфейс, реализаторы которого даже не связаны иерархически. Но это может быть и один конкретный класс. Не забывайте, что всегда можно встретить препятствия на пути реализации всего этого из-за отсутствия подобного полиморфизма в технологии СУБД.
- *Извлекайте преимущества из независимости от клиента.* Реализация ХРАНИЛИЩА предоставляет значительно больше свободы для внесения изменений, чем тот случай, когда клиент вызывает механизмы напрямую. Этим можно воспользоваться для оптимизации быстродействия, варьируя запросы или кэшируя объекты в памяти, по желанию меняя общую методику хранения и поддержания целостности объектов. Можно также облегчить тестирование клиентского кода и объектов модели предметной области, построив легко управляемую симуляцию ХРАНИЛИЩА с хранением объектов в оперативной памяти.

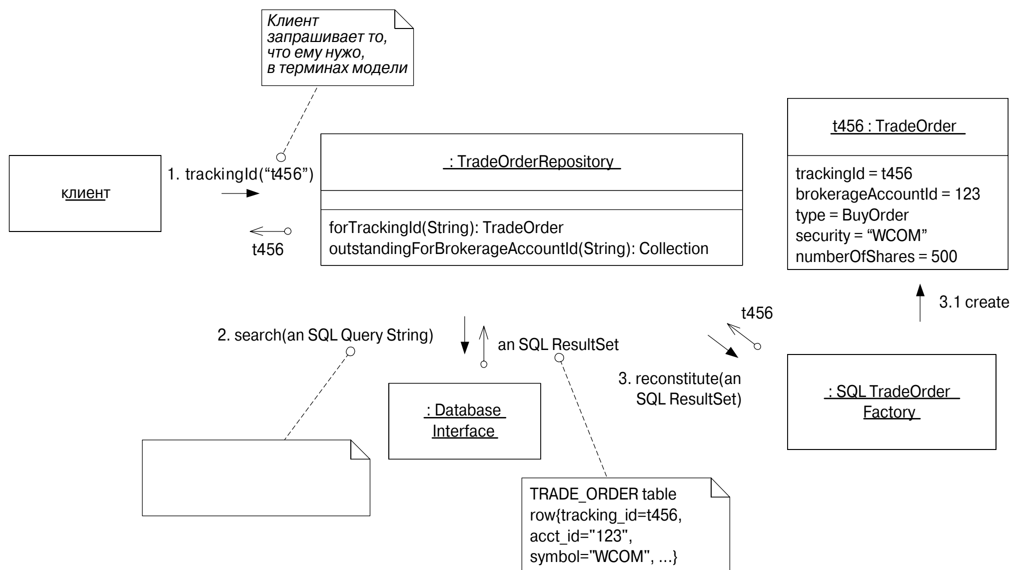


*Рис. 6.21. Инкапсуляция реальной системы хранения данных в объекте-ХРАНИЛИЩЕ*

- *Оставьте контроль транзакций клиенту.* Хотя именно ХРАНИЛИЩЕ помещает данные в базу и извлекает их оттуда, оно, как правило, не должно контролировать их завершение (т.е. выполнять фиксацию транзакции). Конечно, есть искушение, например, зафиксировать транзакцию после сохранения данных, но у клиента на-

верняка есть собственный контекст для корректной инициализации и завершения отдельных рабочих операций. Контроль транзакций со стороны клиента значительно облегчается, если ХРАНИЛИЩЕ в это дело не вмешивается.

Обычно разработчики размещают архитектурную среду для поддержки ХРАНИЛИЩ на инфраструктурном уровне. Кроме обеспечения связи с компонентами инфраструктуры, которые расположены ниже, надкласс ХРАНИЛИЩА может также реализовать некоторые важнейшие запросы, особенно когда имеется механизм гибкого их построения. К сожалению, в такой системе типов, как у Java, приходится типизировать возвращаемые объекты, как "Объекты", оставляя клиенту работу по приведению их к объявленному типу ХРАНИЛИЩА. Но это, конечно, приходится делать с запросами, которые в Java и так возвращают коллекции.

Некоторые дополнительные рекомендации по программированию хранилищ и технические шаблоны для их поддержки (такие, как "объект-запрос", *query object*) можно найти в [13].

## Работа в рамках архитектурной среды

Прежде чем программировать что-нибудь вроде ХРАНИЛИЩА, необходимо хорошенько обдумать инфраструктуру, с которой приходится работать, — особенно архитектурную среду, если она есть. Можно обнаружить, что среда предоставляет такие возможности, с помощью которых легко построить ХРАНИЛИЩЕ, а бывает и так, что она только мешает, и чем дальше, тем больше. Может оказаться, что в архитектурной среде уже определены адекватные шаблоны для поддержания существования объектов, но иногда получается так, что готовые шаблоны вообще ничем не похожи на ХРАНИЛИЩА.

Пусть, например, проект строится на основе J2EE. Если поискать концептуальное сходство между этой средой и ПРОЕКТИРОВАНИЕМ ПО МОДЕЛИ (и при этом помнить, что объект Java Bean и объект-СУЩНОСТЬ — не одно и то же[2]), то можно, например, волевым решением назначить объекты Java Bean корневыми объектами АГРЕГАТОВ. Конструкция в архитектурной среде J2EE, обеспечивающая доступ к таким объектам, называется *EJB Home*. Попытка выдать ее за ХРАНИЛИЩЕ может вызвать и другие проблемы.

В целом, можно посоветовать "не плыть против течения" архитектурной среды. Пытайтесь придерживаться принципов DDD и при этом не отвлекаться на частности, когда среда работает против вас. Ищите сходство между концепциями предметно-ориентированного проектирования и принципами устройства среды, в которой работаете. Все это, конечно, справедливо в предположении, что вы не имеете права уклониться от работы со средой. Во многих проектах на основе J2EE объекты Java Bean вообще не используются. Если у вас есть свобода выбора, работайте с теми средами или их фрагментами, которые согласуются с принятым вами архитектурным стилем.

## Связь с фабриками

ФАБРИКА ведает началом существования объекта, а ХРАНИЛИЩЕ помогает работать с ним в середине и конце его жизни. Если объекты находятся в оперативной памяти или хранятся в объектной базе данных, то тут все просто. Но обычно хотя бы часть данных программы сохраняется в реляционной базе, файле и других необъектных системах. В таких случаях извлеченные из этих мест хранения данные приходится восстанавливать в объектную форму.

[2] Т.е. соответственно, *entity bean* и просто *entity*. Против этой путаницы и предостерегает автор. — *Примеч. перев.*

Поскольку в этом случае ХРАНИЛИЩЕ фактически создает объекты по имеющимся данным, многие считают, что ХРАНИЛИЩА — и есть ФАБРИКИ. С технической точки зрения так оно и есть. Но все-таки полезно на первом плане держать концептуальную модель, а с ее позиций, как уже говорилось, восстановление хранимого объекта не есть создание нового. В методике проектирования, основанной на предметной области, ФАБРИКИ и ХРАНИЛИЩА выполняют разные функции. ФАБРИКА создает новые объекты; ХРАНИЛИЩЕ находит и извлекает старые. ХРАНИЛИЩЕ должно давать клиентам иллюзию, что объекты хранятся прямо в памяти. Бывает, что объекты приходится восстанавливать (да, и при этом создавать новые экземпляры), но концептуально это те же самые объекты, которые уже существовали — это просто середина их жизненного цикла.

Чтобы примирить разные точки зрения, достаточно сделать так, чтобы ХРАНИЛИЩЕ *делегировало* создание объектов ФАБРИКЕ, которая также (теоретически, хотя на практике и редко) могла бы создавать и совсем новые объекты "с нуля".

Четкое разделение этих обязанностей помогает также снять с ФАБРИКИ всякую ответственность за поддержание целостности (непрерывности существования) объекта. Работа ФАБРИКИ — создать объект любой требуемой сложности на основе данных. Если в результате получается новый объект, об этом должен знать клиент, который при желании может добавить его в ХРАНИЛИЩЕ, а оно уже инкапсулирует операции по сохранению объекта в базе данных.

*Рис. 6.22. ХРАНИЛИЩЕ восстанавливает существующий объект с помощью ФАБРИКИ*

*Рис. 6.23. Помещение нового объекта в ХРАНИЛИЩЕ*

Искушение объединить ФАБРИКУ И ХРАНИЛИЩЕ появляется еще в одном случае — при желании реализовать функцию “поиска или создания”. При этом клиент описывает, какой объект ему нужен, и если поиск показывает, что такого объекта еще не существует, то он создается и предоставляется клиенту. Подобных функций следует избегать. В лучшем случае ее наличие создает совсем небольшие удобства, но даже кажущаяся ее полезность исчезает вовсе, если в программе делается различие между СУЩНОСТЯМИ И ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ. Если клиенту нужен ОБЪЕКТ-ЗНАЧЕНИЕ, он обращается к фабрике и получает новый. Как правило, различие между новым и уже существующим объектами играет важную роль в предметной области, и если средства архитектурной среды позволяют сымитировать отсутствие такого различия, то на самом деле они только запутывают дело.

## Проектирование объектов для реляционной базы данных

Самой распространенной необъектной компонентой программных систем, которые в основном следуют объектно-ориентированному подходу, является реляционная база данных. Ее наличие порождает проблемы смешения парадигм (см. главу 5). Но база данных более тесно связана с объектной моделью, чем большинство прочих компонентов. База данных не просто имеет дело с объектами — она хранит в себе постоянную форму тех данных, которые образуют эти самые объекты. Уже немало написано о технических трудностях и особенностях проекции (отображения) объектов на реляционные базы данных, эффективного их хранения и извлечения. В частности, этот вопрос рассматривается в книге [13]. Существуют достаточно отлаженные средства для построения соответствий между этими двумя формами данных и управления ими. Не считая технических трудностей, некорректное построение такого соответствия может иметь существенное влияние и на саму объектную модель.

Наиболее распространены три случая.

1. База данных является в основном хранилищем объектов.
2. База данных была разработана для другой системы.
3. База данных разработана для этой системы, но выступает в роли, отличной от хранилища объектов.

Если структура базы данных специально проектируется для хранения объектов, то стоит и потерпеть некоторые ограничения в модели ради простоты соответствия объектов. Если нет других требований к структуре базы, ее можно спроектировать так, чтобы легче и эффективнее было поддерживать ее агрегатную целостность в процессе обновления. Вообще-то, структура таблиц реляционной базы данных не обязана отражать модель предметной области. Средства отображения данных сами по себе достаточно богаты возможностями, чтобы сгладить любые существенные отличия. Проблема в том, что иметь много накладывающихся друг на друга моделей не очень-то удобно и слишком сложно. К этому случаю применима та рекомендация, которую мы упоминали в разговоре о преимуществах ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ — избегать разделения двух процессов, анализа проблемы и проектирования модели. Да, для этого требуется частично пожертвовать сложностью объектной модели, а иногда пойти на компромисс и в структуре базы данных (например, избирательно применить денормализацию), но без этого есть риск потерять тесную связь между моделью и программной реализацией. Этот подход не требует упрощенного отображения “один объект-одна таблица”. В зависимости от возможностей имеющихся средств отображения данных, возможно агрегирование и комбинирование объектов. Но очень важно, чтобы отображение сохраняло про-

зрачный характер — чтобы его можно было легко понять из чтения кода или названий отображаемых ячеек данных.

- Если база данных выступает в основном хранилищем объектов, не позволяйте модели данных и объектной модели "расходиться слишком далеко", вне зависимости от богатства средств отображения. Пожертвуйте частью отношений между объектами ради того, чтобы сделать модель ближе к реляционной. Не бойтесь применять такие формально реляционные стандарты, как нормализация, если это помогает в отображении данных.
- Процессы, протекающие вне объектной системы, вообще не должны иметь доступа к хранилищу объектов, потому что они могут нарушить накладываемые объектами инвариантные ограничения. Кроме того, предоставление им права доступа заблокирует модель данных от изменений, и это еще скажется, когда придет время рефакторинга.

С другой стороны, во многих случаях данные поступают из устаревшей или внешней системы, которая никогда и не задумывалась как хранилище объектов. В такой ситуации в одной системе фактически соседствуют две модели предметной области. Этот вопрос подробно разбирается в главе 14. Иногда имеет смысл приспособиться к модели, принятой в другой системе, а иногда, наоборот, — сделать свою модель совершенно другой.

Еще одна причина, по которой приходится делать исключения — это быстродействие. Для решения проблем в этой области программисту приходится идти на многие ухищрения.

Но для важного и распространенного случая, когда реляционная база данных служит постоянным хранилищем объектов из объектно-ориентированной предметной области, лучше всего применять самый прямой подход. Строка таблицы должна содержать объект — возможно, вместе с его подобъектами в виде АГРЕГАТА. Внешний ключ в таблице должен транслироваться в ссылку на другой объект-СУЩНОСТЬ. Если и встречается необходимость иногда отойти от этого прямого подхода, это не должно приводить к полному забвению принципа прямого соответствия.

Привязать объектную и реляционную составляющую к одной и той же модели помогает ЕДИНЫЙ ЯЗЫК. Имена и ассоциации элементов в объектах должны до мелочей соответствовать именам и ассоциациям в реляционных таблицах. Хотя в присутствии мощных средств отображения данных это может показаться несущественным, даже небольшие различия в отношениях между данными могут вызвать большую путаницу.

Традиция рефакторинга, которая все больше овладевает объектно-ориентированным миром, пока не слишком сильно повлияла на проектирование реляционных баз данных. Более того, серьезные проблемы переноса данных делают частые изменения нежелательными. Это может затормозить рефакторинг объектной модели, но если модель базы данных и объектная модель начинают расходиться, то может быстро потеряться прозрачность, наглядность преобразования данных.

Наконец, могут быть и причины для введения такой структуры базы данных, которая решительно отличается от объектной модели, пусть даже база специально создавалась именно для данной программной системы. База данных может также использоваться другой программой, в которой вообще не инициализируются экземпляры объектов. Такая база может практически не требовать изменений даже тогда, когда поведение объектов быстро меняется. Тогда возникает искушение углубить разрыв между системой и базой данных. Часто это делается непреднамеренно — разработчикам просто не удается вести базу данных "в ногу" с моделью. Если же такой разрыв выбирается сознательно, в результате вполне может получиться аккуратная и экономная структура таблиц базы, а не корявое нечто, порожденное многочисленными попытками привязать базу данных к самой последней версии объектной модели.