Глава 4

Новая используемая по умолчанию архитектура

В прошлом я разрабатывал для своих приложений архитектуры, ориентированные, в основном, на данные. Затем я постепенно изменил свой подход, и теперь используемые мной по умолчанию архитектуры ориентированы на модель предметной области. Отсюда и название этой главы.

Вы можете спросить: “А для кого эта архитектура новая?” Все зависит от обстоятельств. Так, если описать модель предметной области посторонним людям, они спросят: “А это всегда так делается?” В то же время немало разработчиков зададут вопрос: “А это действительно работает?”

Независимо от того, к какой из двух категорий людей вы относитесь, мы пойдем вместе по пути исследования модели предметной области и, в частности, ее построения по принципам *проектирования на основе предметной области* (ППО).

Этот путь мы начнем с краткого изложения моделей предметной области и ППО. На конкретном примере мы рассмотрим перечень требований к приложению и наметим возможные решения, направленные на реализацию разных свойств этого приложения. А для того чтобы получить более ясное представление о требованиях, мы рассмотрим их под новым углом зрения и набросаем черновой вариант исходного пользовательского интерфейса. В конце этой главы мы коснемся стилей исполнения моделей предметной области.

Итак, приступим.

Основа новой используемой по умолчанию архитектуры

Как упоминалось выше, новая используемая по умолчанию архитектура основана на модели предметной области. Эту модель мы уже обсуждали в главе 1, а здесь рассмотрим ее более подробно.

На заметку

Поль Гиленс сделал следующее замечание относительно слова “используемая по умолчанию” в названии этой главы: у читателей может сложиться впечатление, будто такой подход применяется ко всем разрабатываемым системам. Следует, однако, подчеркнуть, что это совсем не так. Данная книга отнюдь не предназначена в качестве образца для подражания. Ведь главный принцип ППО — предоставить прикладной задаче самой определять окончательное решение!

Под “моделью предметной области” в контексте архитектуры приложений обычно подразумевается применение шаблона Domain Model [Fowler PoEAA]. Это сродни попытке, характерной для традиционного направления объектной ориентации, как можно точнее перенести упрощенное (а вернее, абстрактное) представление действительности из предметной области в объектно-ориентированную модель. Получаемый в итоге код очень точно соответствует выбранному уровню абстракции. Это относится к состоянию и поведению, а также к возможным путям перемещения и отношениям между объектами.

На заметку

Кто-то (к сожалению, не помню кто) назвал применение шаблона Domain Model в C# “программированием на C#, как на SmallTalk”. Это было, на мой взгляд, сказано в весьма положительном смысле, хотя для некоторых такой положительный смысл поначалу не совсем очевиден.

Разумеется, модель предметной области — не панацея, но у нее, безусловно, имеется ряд положительных свойств, что особенно справедливо для крупномасштабных и сложных приложений с длительным сроком службы. По мнению Мартина Фаулера [Fowler PoEAA], если попробовать применить модель предметной области в одном проекте, то в дальнейшем, скорее всего, возникнет желание использовать ее даже в небольших, простых приложениях.

Как определить долговечность приложения

Когда речь заходит о приложениях с длительным сроком службы, я припоминаю одно приложение, которое меня попросили создать. Поставщик услуг глобальной связи для передачи данных предложил мне разработать черновой вариант приложения справочной службы для нового бюро в Швеции. Сроки были крайне сжатыми — приложение должно было работать уже на следующий день, чтобы сотрудники не сидели в этом бюро, оснащенном по последнему слову

техники, с листом бумаги и ручкой в руках, когда пресса пожалует к открытию. Это приложение все равно собирались заменить на стандартное для данной компании приложение через несколько недель, и поэтому мне сказали, чтобы я не особенно старался разрабатывать такое приложение основательно.

Да, вы угадали. Это было приложение, сделанное на скорую руку, хотя оно эксплуатировалось в течение пяти лет. Разумеется, я значительно усовершенствовал его со своими коллегами, но в то же время я не смог убедить заказчика перейти на другую платформу, более подходящую для столь ответственного приложения.

Смещение акцента с базы данных на модель предметной области

Я лично сместил свой акцент с базы данных на модель предметной области, главным образом, потому, что стал уделять больше внимания не столько эффективности, сколько сопровождению приложений. Этот новый стиль проектирования совсем не означает, что я принес в жертву эффективность, но я стараюсь избегать преждевременной оптимизации каждой мелкой детали, что нередко обходится дороже, чем оно того заслуживает. При этом я пытаюсь построить как можно более ясную модель предметной области, что в конечном итоге упрощает оптимизацию, когда она действительно требуется. Нельзя сказать, что я совсем забыл о базе данных — просто она не находится в главном фокусе моего внимания. Я стараюсь находить разумные компромиссы и считаю, что с базой данных следует обращаться очень аккуратно.

Проектирование, ориентированное на модель предметной области, предоставляет больше возможностей для сопровождения приложений в долгосрочной перспективе, поскольку оно оказывается более ясным, а реализация точнее соответствует уровню абстракции модели. Еще одна важная причина для такого метода проектирования заключается в том, что эффективная модель предметной области служит удобным инструментом для сокращения дублирования логики. (Разумеется, все эти свойства присущи и объектной ориентации. Я рассматриваю модель предметной области в качестве стиля проектирования, где объектная ориентация применяется в чистом виде, раздвигая границы для написания удобно сопровождаемого и, конечно, проверяемого кода.)

Возможно, шаблон Domain Model [Fowler PoEAA] означает для вас нечто совершенно особенное, но ведь существует множество вариантов его применения. Назовем эти варианты разными стилями проектирования.

Более конкретный акцент на ППО

Стиль построения модели предметной области приводит нас в конечном итоге к ППО [Evans DDD].

Для ППО имеются самые разные средства (см. подробное описание в главе 1). В частности, это набор шаблонов, помогающих структурировать модель предметной области. И вскоре мы начнем пользоваться этими инструментами.

Но прежде чем переходить к более конкретным вопросам, рассмотрим вкратце процесс разделения на слои в соответствии с ППО.

Разделение на слои в соответствии с ППО

В своей предыдущей книге [Nilsson NED] я отвел немало места строгой схеме разделения на слои, но здесь все же нужно сказать несколько слов о том, как я теперь представляю себе разделение на слои.

С одной стороны, я делаю большую ставку на разделение на слои. А поскольку я истово исповедую ППО, то стараюсь всеми силами выделить инфраструктуру, в том числе и сохраняемость, в отдельный слой — подальше он базового слоя предметной области.

С другой стороны, я меньше думаю о разделении на слои и уделяю больше внимания только одному: слою предметной области. При этом я не разделяю данный слой (т.е. модель предметной области) на более мелкие слои, сохраняя его на уровне довольно крупных структурных единиц. (На самом деле некоторые шаблоны ППО вполне подходят для задач, решать которые раньше приходилось с помощью разделения на слои.)

До сих пор мы рассмотрели два слоя: инфраструктуры и предметной области. Поверх этих слоев может находиться слой приложения, обеспечивая определенное согласование, но он очень тонкий и только поручает работу слою предметной области (см. шаблон Service Layer (Уровень обслуживания) [Fowler PoEAA], который перекладывает всю работу на модель предметной области подобно классам сценария.)

И наконец, на самом верху располагается слой пользовательского интерфейса.

Пример упрощенной схемы разделения на слои приведен на рис. 4.1.

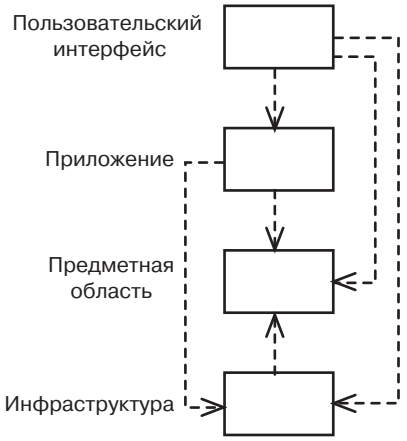

Рис. 4.1. Разделение на слои, типичное для проектов, выполняемых в стиле ППО

На заметку

Опять же, все сказанное выше о разделении на слои нельзя автоматически принимать за эталон только потому, что об этом написано в данной книге! Каждое решение следует подвергать критическому анализу!

А теперь мне бы хотелось сказать несколько слов о том, как я пользуюсь разделением на слои в сравнении со своим старым стилем проектирования. Прежде всего, слой приложения не обязателен — он нужен лишь в том случае, если приносит какую-то пользу. А поскольку слой предметной области получается у меня теперь более обогащенным, чем прежде, то слой приложения меня зачастую мало интересует.

Еще одно отличие заключается в следующем: вместо того чтобы передавать все вызовы вниз по иерархии, слой инфраструктуры может "знать" о слое предметной области, и в нем могут создаваться экземпляры объектов при восстановлении из состояния сохраняемости. Самое главное — чтобы модели предметной области не было ничего известно об инфраструктуре.

На заметку

Следует также подчеркнуть, что в данном случае не принимается во внимание *разбиение*, или рассечение на части в другом измерении по сравнению с разделением на слои. Это очень важно для крупных приложений. Мы еще вернемся к этому вопросу в главе 10.

А теперь пора отправляться в путь, чтобы опробовать некоторые концепции, рассмотренные в этой и предыдущих главах, с акцентом на слое предметной области. И для этого лучше всего подыскать подходящий пример.

Первый набросок

Для большей конкретности составим перечень задач и возможностей приложения, чтобы посмотреть, как в процессе его проектирования решаются некоторые общие задачи упорядочения приложения.

Разумеется, работа над списком возможностей при проектировании на основе предметной области зависит от многих факторов. Как упоминалось выше, рассуждения об "используемой по умолчанию архитектуре" могут показаться странными из-за того, что архитектура в первую очередь зависит от решаемой задачи.

Тем не менее мне бы хотелось рассмотреть здесь одно из возможных решений. Пока что оно будет иметь весьма общий, эскизный характер, а о его деталях речь пойдет в последующих главах книги. Это будет лишь первая попытка сделать грубый набросок проекта, который в дальнейшем обрастет деталями.

Задачи и возможности для примера предметной области

Приведенный ниже список требований будет использован в последующих главах книги для обсуждения некоторых концепций. Итак, рассмотрим подробнее каждую задачу или возможность в произвольном порядке.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра.
   Персоналу, обслуживающему клиентов, необходимо предоставить возможность для удобного поиска клиентов. В частности, они должны пользоваться метасимволами в многочисленных полях, включая имя, адрес, ответственное лицо и т.д. Кроме того, у них должна быть возможность искать клиентов с заказами определенного типа, определенной величины, на определенные виды продукции и т.д. Таким образом, здесь речь идет о полноценной утилите поиска, которая в итоге формирует список клиентов со своими номерами, именами и фамилиями, адресами.
2. Список заказов, формируемый при просмотре отдельного клиента.
   В списке должна отображаться общая стоимость заказа, а также его состояние, тип, дата составления, имя и фамилия ответственного лица.
3. Заказ может состоять из самых разных строк.
   Заказ может быть многострочным, причем в каждой строке описывается заказанная продукция и ее номер.
4. Большое значение имеет выявление конфликтов параллельной обработки.
   В данном случае достаточно выбрать оптимистический вариант управления параллельным выполнением операций, т.е. считать вполне приемлемым, когда пользователя уведомляют о конфликте с предыдущим результатом сохранения после того, как он завершит свою работу и попытается сохранить ее. При этом конфликтами считаются только те, что приводят к настоящим противоречиям. Следовательно, в данном решении следует предусмотреть блок управления версиями для клиентов и заказов. (Это может оказать некоторое влияние на ряд других возможностей.)
5. У клиента может быть долг, не превышающий определенную суму.
   У каждого клиента этот лимит свой. Он устанавливается при первоначальном вводе клиента, а впоследствии может быть изменен. Наличие неоплаченных заказов на общую сумму, превышающую данный лимит, считается противоречием. Это противоречие можно допустить в том случае, если пользователь уменьшит лимит. Пользователь, уменьшивший лимит, получает соответствующее уведомление, но операция сохранения разрешается. Тем не менее заказ не может быть введен или изменен с превышением лимита.
6. Заказ не может быть составлен на сумму свыше 1 млн крон (в данном примере используется шведская валюта).

Этот лимит, в отличие от предыдущего, является правилом, распространяющимся на всю систему.

7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер.

   Допускается следование номеров не по порядку.

8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении.

   Это означает, что лимит, рассматривавшийся в п. 5 данного списка и установленный для клиента, будет проверен на обоснованность.

9. У заказа должен быть клиент, а у строки заказа — заказ.

   Заказов с неопределенными клиентами быть не должно. Это же относится и к строкам заказа — они должны принадлежать определенному заказу.

10. Сохранение заказа и его строк должно быть атомарным.

    Откровенно говоря, я не уверен в том, что такая возможность действительно нужна. Ведь вполне допустимо, чтобы строки заказа вводились после его первоначального составления. Но мне требуется какое-то правило для данного примера, чтобы иметь возможность, связанную с защитой транзакций.

11. У заказов должно быть состояние подтверждения, изменяемое пользователем.

    Это состояние может изменяться пользователями при переходе от подтверждения заказа к неподтверждению и обратно. Что же касается других состояний, то их смена должна происходить косвенно другими методами, доступными в модели предметной области.

Итак, мы составили аккуратный, простой список возможностей, чтобы пользоваться им при рассмотрении разных решений в общей перспективе.

На заметку

Приведенный выше список возможностей может показаться слишком ориентированным на данные. В некоторых случаях такая кажущаяся ориентация на данные воплощается на практике в свойства с некоторым поведением.

Следует, однако, заметить, что мы стремимся применить модель предметной области. И в этом случае главная трудность заключается в следующем: как нам обращаться с данными, если мы пользуемся реляционной базой данных. Поэтому, когда дело доходит до объектной ориентации, имеет смысл уделить данным чуть больше внимания, чем обычно.

Итак, сформулировав возможности и задачи, перейдем к рассмотрению возможного их решения.

Решение задач по очереди

А теперь приступим к решению (как обычно, по очереди) задач из приведенного выше списка задач и возможностей и посмотрим, к чему это приведет. Сосредоточив основное внимание на модели предметной области, попробуем уловить и сформировать универсальный язык, не отвлекаясь на инфраструктуру или другие слои.

Впрочем, не помешает составить некоторое представление и о технической среде. Попробуем не усложнять ее и выберем в качестве контекста приложение с графическим пользовательским интерфейсом, имеющим широкие функциональные возможности (Rich GUI). Это приложение должно быть реализовано в настольном исполнении на стороне клиента и объединено с вместе с моделью предметной области и физическим сервером реляционной базы данных в одну локальную сеть.

> ***На заметку***
>
> Ниже упоминаются многие шаблоны, но мы еще не раз вернемся к ним в последующих главах. Полный перечень шаблонов приведен в Приложении Б.

Итак, рассмотрим поставленные задачи по порядку.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра

Первое требование имеет непосредственное отношение к данным и, отчасти, к поисковому поведению. Прежде всего, наметим структуру класса `Customer` и его окружения (пока что без класса `Order`; рис. 4.2).

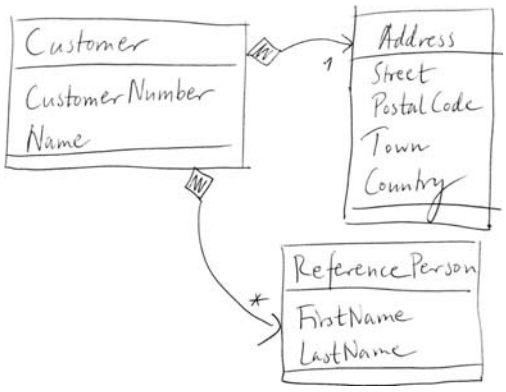

Рис. 4.2. Класс `Customer` и его окружение

На заметку

Как ни странно, но для создания рисунков от руки имеются достаточно веские основания. С их помощью удобно показывать первые наброски, самые общие идеи, с которых, на мой взгляд, можно начинать проект. А код — это лишь исполняемый, отчетливый артефакт, хотя он и является самым важным представлением настоящей модели.

Как видите, я выбрал для клиента достаточно простой (наивный) класс `Customer`, состоящий из классов адреса (`Address`) и ответственных лиц (`ReferencePerson`). Я решил не вносить подобного рода поисковое поведение непосредственно в класс `Customer`, поэтому на данный момент мы располагаем только данными о классах, приведенных на рис. 4.2.

На заметку

Вместо этого можно было бы воспользоваться шаблоном Party Archetype (Групповой архетип) [Arlow/Neustadt Archetype Patterns] или же другими видами реализации ролей, но пока что мы будем пользоваться простыми структурами.

Первое требование можно было бы выполнить, используя шаблон Query Object (Объект запроса) [Fowler PoEAA], о котором речь уже шла в главе 2, но на практике оказывается, что применение таких шаблонов лучше сделать немного инкапсулированным, если имеется такая возможность. Ведь зачастую приходится определять немало из того, что касается результатов запросов, но не имеет никакого отношения к критериям поиска. Речь идет о порядке сортировки, других скрытых критериях, оптимизации, месте, куда следует отправлять объект запроса для последующего выполнения, и прочем. Поэтому я предпочитаю пользоваться шаблоном Repository (Хранилище) [Evans DDD], чтобы немного инкапсулировать выполнение запроса. Это позволяет также сделать код менее многословным для потребителей, а программирование интерфейса API — более ясным.

В главе 2 мы рассматривали шаблон Factory (Фабрика) [Evans DDD] и выяснили, что он определяет начало жизненного цикла экземпляра объекта. А за остальную часть жизненного цикла после создания экземпляра и до его "кончины" отвечает шаблон Repository. В частности, шаблон Repository наводит мост между базой данных и моделью предметной области, когда требуется получить экземпляр объекта, который прежде был сохраняемым. При этом шаблон Repository использует инфраструктуру для выполнения своей задачи.

Методу из шаблона Repository можно было бы передать целый список параметров: по одному на каждое поле фильтра. Но это, без сомнения, привело бы к недоб-

рокачественному коду, который оказался бы не совсем ясным, а значит, и неудобным для сопровождения. Более того, для шаблона Repository не совсем ясно, что делать с параметром `Name` строкового типа, особенно если строка пуста. Означает ли это, что мы должны искать клиентов с пустыми именами или же нас вообще не интересуют их имена? Конечно, можно составить волшебную строку, которая означала бы, что мы ищем клиентов с пустыми именами, а еще лучше — вынудить пользователя вводить метасимвол, если его не интересует имя клиента. Но ни одно из этих решений нельзя считать вполне пригодным. С другой стороны, можно было бы воспользоваться шаблоном Query Object [Fowler PoEAA], в особенности специальным для предметной области, или же шаблоном Specification (Описание) [Evans DDD], но и это решение оказалось бы, на мой взгляд, слишком поспешным и опрометчивым. Поэтому выберем самое простое решение и рассмотрим пока что лишь два возможных критерия. Пример такого решения приведен на рис. 4.3.

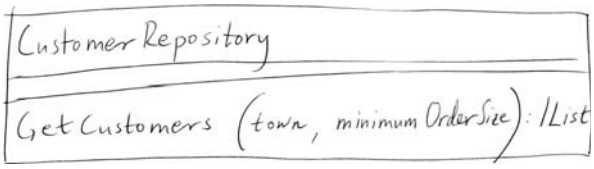

Рис. 4.3. Класс `CustomerRepository` для гибких критериев

На заметку

Разумеется, внутри шаблона Repository можно было бы использовать шаблон Query Object.

ППО прекрасно сочетается с РПТ — вы получаете мгновенную реакцию на свои действия и опробуете модель с помощью тестового кода, что позволяет вам лучше ее понять. Но пока что мы воздержимся от следования по пути РПТ и исследуем набросок модели с помощью мелких тестов только для того, чтобы рассмотреть замысел данного решения под другим углом зрения. В следующей главе мы вернемся к этому вопросу и попробуем разработать модель более тщательно, точнее придерживаясь принципов РПТ.

Итак, для того чтобы выбрать всех клиентов в определенном городе (Роннебю — Ronneby) хотя бы с одним заказом на сумму более 1 тыс. шведских крон, воспользуемся следующим тестом:

```
[Test]
public void
  CanGetCustomersInSpecificTownWithOrdersOfCertainSize()
{
```

```
    int numberOfInstancesBefore = _repository.GetCustomers
      ("Ronneby", 1000).Count;

    _CreateACustomerAndAnOrder("RonnebyЭ, 20000);

    Assert.AreEqual(numberOfInstancesBefore + 1
      , _repository.GetCustomers ("Ronneby", 1000).Count);
}
```

В данном примере метод `GetCustomers()` из шаблона Repository используется для выборки всех клиентов, удовлетворяющих критериям поиска города и минимальной величины заказа.

2. Список заказов, формируемый при просмотре отдельного клиента

Между классами `Customer` и `Order` можно было бы установить двунаправленную связь, когда каждому клиенту соответствует список заказов, а каждому заказу — один клиент. Но такая двунаправленность обходится дорого в смысле сложности, тесного связывания и дополнительных издержек. Поэтому было бы неплохо, на мой взгляд, организовать доступ к заказам клиента через класс хранилища заказов (`OrderRepository`). Это приводит нас к модели, представленной на рис. 4.4.

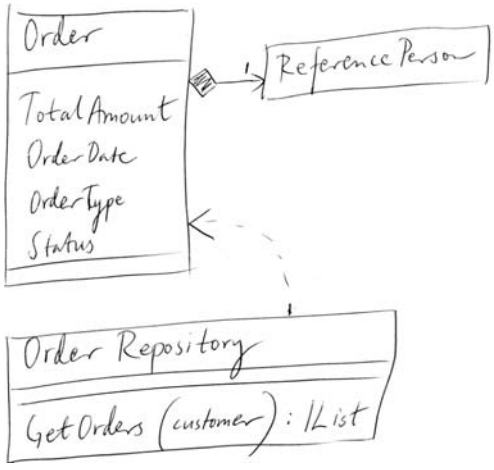

Рис. 4.4. Классы `OrderRepository`, `Order` и их окружение

Так, если потребителю требуется посмотреть заказы определенного клиента, он должен обратиться к классу `OrderRepository` следующим образом:

```
[Test]
public void CanGetOrdersForCustomer()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
```

```
    ("Ronneby", 20000);

  IList ordersForTheNewCustomer =
    _repository.GetOrders(newCustomer);

  Assert.AreEqual(1, ordersForTheNewCustomer.Count);
}
```

Классу `Customer` можно было бы присвоить свойство `OrderList` (Список заказов) и обращаться к хранилищу неявно, но, на мой взгляд, это лучше делать явно. Кроме того, следует избегать связывания классов `Customer` и `OrderRepository`.

Отдельного упоминания требует общая сумма каждого заказа. Это более сложный вопрос, чем кажется на первый взгляд, поскольку для расчета общей суммы требуются все строки заказа. Конечно, это не такая уж непреодолимая трудность, но она все же настораживает. Возможно, мои опасения преждевременны, но я не могу ничего с этим поделать, поскольку загрузка строк заказа может обойтись недешево, когда потребуется, например, показать довольно внушительный список заказов определенного клиента. Поэтому если строки заказа не загружаются, для общей суммы (свойства `TotalAmount`) лучше использовать простое поле, сохраняемое в таблице заказов (`Orders`). Если же строки заказа загружаются, то вместо свойства `TotalAmount` используется полный расчет общей суммы заказа.

Конечно, это не самый актуальный вопрос на данный момент. Но он решается для потребителей очень просто следующим образом:

```
[Test]
public void CanGetTotalAmountForOrder()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 420);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  Assert.AreEqual(420, newOrder.TotalAmount);
}
```

Появление этих строк кода объясняется тем, что мы имеем дело с агрегатом данных, а сами строки отражают внутреннее содержание заказа. Хотя в некоторых ситуациях это может привести к снижению производительности. В таком случае решение можно искать с помощью шаблона Lazy Load (Загрузка по требованию) [Fowler PoEAA], используемого, например, для оперативной загрузки списков из базы данных, или же с помощью представления, оптимизированного “только для чтения”.

Однако, как уже отмечалось, данное решение следует рассматривать в том случае, если простого и прямого решения окажется недостаточно.

3. Заказ может состоять из самых разных строк

Эта возможность реализуется достаточно просто. На рис. 4.5 представлена улучшенная модель.

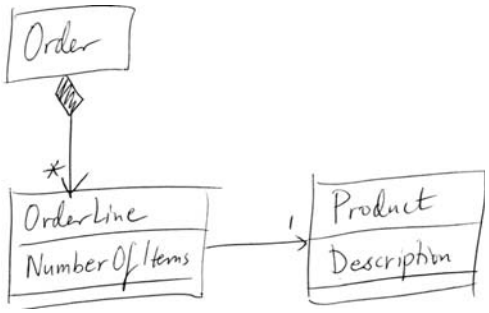

Рис. 4.5. Заказ (класс `Order`) и его строки (класс `OrderLine`)

Как видите, я рассматриваю в этой модели и однонаправленную связь. Конечно, можно было бы обойтись отправкой строки заказа (`OrderLine`) вместе с ее заказом (`Order`), если бы знать, что и то и другое понадобится для определенной части логики. Впрочем, риск отправки заказа вместе со строкой из другого заказа не очень велик.

```
[Test]
public void CanIterateOverOrderLines()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 420);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  foreach (OrderLine orderLine in newOrder.OrderLines)
    return;

  Assert.Fail("Этого у меня пока что не должно быть");
}
```

Обратите также внимание на заметные отличия в модели, приведенной на рис. 4.5, по сравнению с реляционной моделью, описывающей то же самое. Для реляционной модели были бы характерны отношения “один ко многим” и “многие к одному” между классами `Product` и `OrderLine`. Но до сих пор нас совершенно не интересовало отношение “один ко многим” в данной конкретной модели предметной области.

4. Большое значение имеет выявление конфликтов параллельной обработки

Эта задача посложнее. На мой взгляд, обоснованное ее решение состоит в отделении класса `Customer` от класса `Order`, который должен составить вместе с классом

OrderLine отдельную структурную единицу для выявления конфликтов параллельной обработки. Это вполне согласуется с применением модели предметной области для логики, необходимой для реализации остальных возможностей приложения. Для этой цели подходит шаблон Aggregate (Агрегат) [Evans DDD], который позволяет определить объекты, принадлежащие определенным группам объектов, с которыми обычно приходится работать как с едиными блоками, загружая их вместе (поначалу хотя бы для написания сценариев), оценивая общие правила для них и т.д.

Следует, однако, заметить, что шаблон Aggregate не относится к самым нужным специальным шаблонам, но чаще применяется для смыслового наполнения модели.

В частности, шаблон Aggregate служит для создания отдельных структурных единиц с целью выявить конфликты параллельной обработки (рис. 4.6).

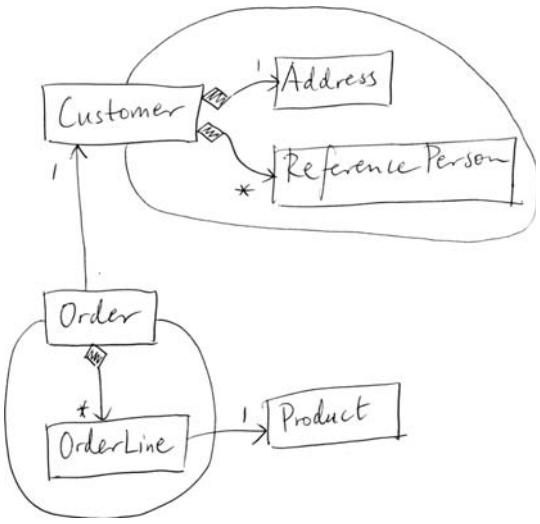

Рис. 4.6. Агрегаты

5. У клиента может быть долг, не превышающий определенную сумму

И эта задача непроста. Первое ее решение, которое приходит на ум, вероятно, состоит в том, чтобы ввести свойство TotalCredit (Общий кредит) в класс Customer. Но такое решение не совсем удачно, поскольку оно настолько прозрачно, что потребитель не увидит расходы при вызове данного свойства. Кроме того, оно не лишено противоречий. Если я даже не вижу, что у агрегата Customer имеются заказы (Order), значит, такую структурную единицу я не могу считать подходящей для обеспечения непротиворечивости в самой модели предметной области. А это служит явным признаком того, что следует искать другое решение.

В модели предметной области, на мой взгляд, лучше применить шаблон Service (Обслуживание) [Evans DDD], который может сообщить текущие сведения об общей сумме кредита для клиента. Соответствующий вид модели приведен на рис. 4.7.

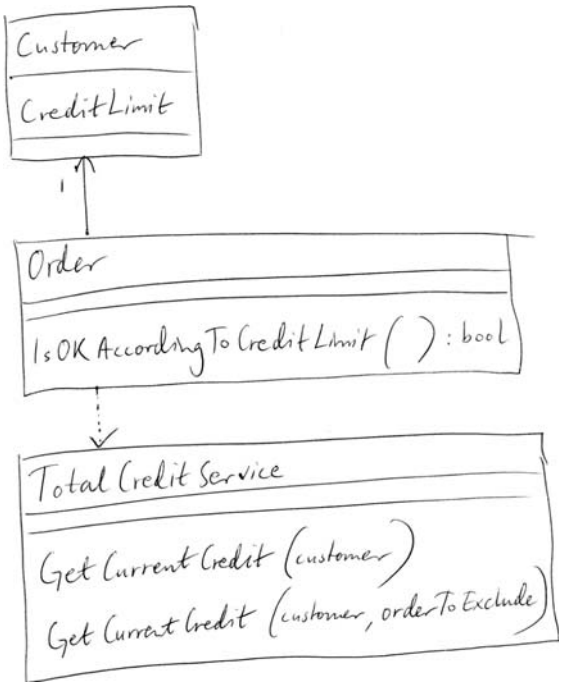

Рис. 4.7. Класс `TotalCreditService`

Основанием для перегрузки метода `GetCurrentCredit()` класса `TotalCreditService` (Обслуживание общего кредита) может, на мой взгляд, служить потребность в проверке величины текущего кредита, не касаясь текущего заказа, который составляется или изменяется. По крайней мере, это неплохая идея. Посмотрим, как такое обслуживание будет выглядеть на деле (не обращая внимания на взаимодействие между отдельными частями модели).

```
[Test]
public void CanGetTotalCreditForCustomerExcludingCurrentOrder()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 22);

  Order secondOrder = _CreateOrder(newCustomer, 110);

  TotalCreditService service = new TotalCreditService();

  Assert.AreEqual(22+110
    , service.GetCurrentCredit(newCustomer));
  Assert.AreEqual(22,
    service.GetCurrentCredit(newCustomer, secondOrder));
}
```

Общая идея взаимодействия классов `Order`, `Customer` и обслуживания должна быть вам ясна из рис. 4.7 и приведенного выше фрагмента кода.

6. Заказ не может быть составлен на сумму свыше 1 млн крон

Возможность выявления конфликтов параллельной обработки была предусмотрена выше в виде отдельной структурной единицы, а точнее агрегата, составленного из классов `Order` и `OrderLine` с помощью шаблона Aggregate, и поэтому мы вправе воспользоваться данной возможностью в модели предметной области. Инвариант агрегата в данном случае состоит в том, что сумма заказа не должна превышать 1 млн крон. Это правило не должны нарушать другие пользователи, чтобы не создавать осложнения при обработке определенного заказа. Ведь иначе у меня или у другого пользователя возникнет конфликт параллельной обработки. Поэтому в классе `Order` следует создать метод `IsOKAccordingToSize()`, который может быть вызван (при желании) потребителем (рис. 4.8).

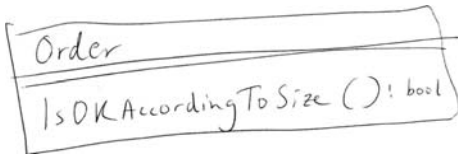

Рис. 4.8. Класс `Order` с методом `IsOKAccordingToSize()`

Ниже приведен простой тест для демонстрации интерфейса API:

```
[Test]
public void CanCheckThatAnOrdersTotalSizeIsOK()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 2000000);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  Assert.IsFalse(newOrder.IsOKAccordingToSize());
}
```

> ***На заметку***
>
> В дальнейшем для данного правила следует рассмотреть возможность применения шаблона Specification [Evans DDD].

7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер

Независимо от моего отношения как разработчика к этому требованию, оно очень важно и типично с точки зрения пользователя. В первом приближении его можно вполне удовлетворить средствами базы данных. Так, в SQL Server для этой цели служат ключи IDENTITY. В самой модели предметной области не нужно ничего делать, кроме обновления объекта-сущности (Entity) после его ввода в таблицу базы данных. А до этого в модели предметной области будут, вероятно, отображаться нулевые значения свойств `OrderNumber` (Номер заказа) и `CustomerNumber` (Номер клиента) (рис. 4.9).

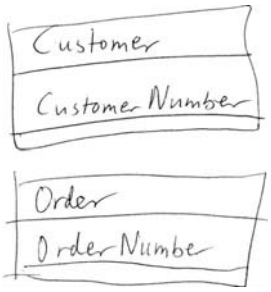

Рис. 4.9. Улучшенные классы `Order` и `Customer`

Я уже жалею, что поторопился с таким решением. Ведь это означает, что номер заказа распределяется при первом же сохранении заказа. Но я не уверен, что это удачная идея.

По существу, я смешал два разных понятия: шаблон Identity Field (Поле идентификации) [Fowler PoEAA], предназначенный для связывания экземпляра объекта со строкой в таблице базы данных (используя первичный ключ из этой строки в виде значения в объекте), и идентификационный номер, имеющий прикладное значение. Иногда эти понятия совпадают, но чаще всего они должны быть представлены двумя разными идентификаторами.

Довольно странным кажется и то, что клиенту присваивается удобный для использования номер в виде маркера, как только он переходит в состояние сохраняемости. Вероятно, этот вариант следовало бы предусмотреть на тот случай, если клиент работает в системе (после проверки его полномочий, а возможно, и подтверждения вручную). К этому вопросу мы еще вернемся в последующих главах книги.

8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении

Для решения этой задачи потребуется еще один шаблон Service [Evans DDD] в модели предметной области, чтобы инкапсулировать обмен информацией с кредитным учреждением (рис. 4.10).

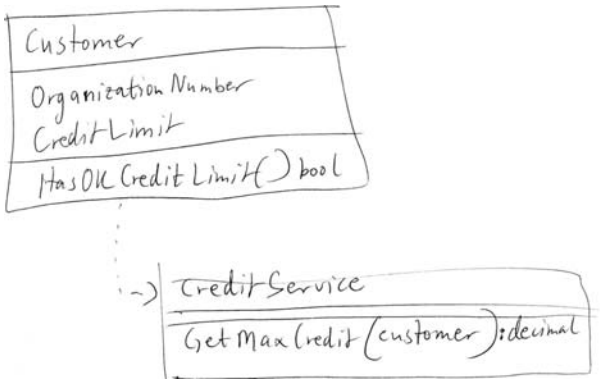

Рис. 4.10. Класс `CreditService`

И в этом случае все сводится к взаимодействию объекта-сущности (в данном случае класса `Customer`) с шаблоном Service, чтобы предоставить этому объекту возможность самому отвечать на поставленный вопрос. Посмотрим, как это взаимодействие происходит на деле в небольшом тесте. (Следует также заметить, что для реального обслуживания клиента, вероятно, потребуется номер его организации (`OrganizationNumber`), который отличается о номера самого клиента (`CustomerNumber`) и является официальным идентификационным номером, предоставляемым органами, регистрирующими организацию.)

```
[Test]
public void CantSetTooHighCreditLimitForCustomer()
{
  Customer newCustomer = _CreateACustomer("Ronneby");

  // Внести упрощенный вариант класса CreditService
  // не допускающий кредит на сумму более 300 крон
  newCustomer.CreditService = new StubCreditService(300);

  newCustomer.CreditLimit = 1000;

  Assert.IsFalse(newCustomer.HasOKCreditLimit);
}
```

На заметку

Грегори Янг обратил внимание на недоброкачественность приведенного выше кода. Это объясняется, скорее всего, неатомарным характером операции. Сначала значение задается, а следовательно, перезаписывается старое значение. Затем это значение проверяется, если оно запоминается. Вместо этого, возможно, лучше было бы использовать выражение `Customer.RequestCreditLimit(1000)`. Мы еще вернемся к этому вопросу в главе 7.

9. У заказа должен быть клиент, а у строки заказа — заказ

Это типичное и вполне обоснованное требование, которое связано с ограничениями в базе данных на целостность ссылочных данных. Теперь нужно проверить также модель предметной области. Нет смысла переводить в состояние сохраняемости изменения, вносимые в модель предметной области, если ясно, что в ней еще немало неточностей, как в данном случае. Но вместо проверки этой модели в качестве первой попытки можно сделать ее обязательной к применению благодаря классу `OrderFactory` (Фабрика заказов) как средству составления заказов, а раз уж приходится иметь дело со строками заказов (`OrderLine`), то аналогичным образом можно сделать для них обязательным наличие заказа (`Order`) и продукции (`Product`) (рис. 4.11).

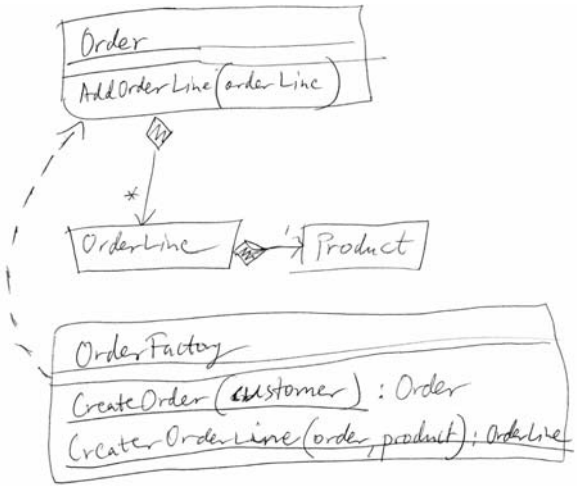

Рис. 4.11. Класс `OrderFactory` и улучшенный класс `Order`

Как обычно, проверим взаимодействие разных частей модели с помощью теста.

```
[Test]
public void CanCreateOrderWithOrderLine()
{
  Customer newCustomer = _CreateACustomer("KarlskronaЭ);

  Order newOrder = OrderFactory.CreateOrder(newCustomer);

  // В классе OrderFactory будет использован метод
  // AddOrderLine() из класса Order.
  OrderFactory.CreateOrderLine(newOrder, new Product());

  Assert.AreEqual(1, newOrder.OrderLines);
}
```

Пожалуй, это похоже на избыточное проектирование, поскольку модель имеет теперь не такой гладкий и аккуратный вид. Впрочем, об этом мы поговорим в следующей главе.

Раз уж мы коснулись класса `OrderFactory`, мне бы хотелось упомянуть об удачной идее использовать шаблоны Null Object (Пустой объект) [Woolf Null Object] для классов `OrderType`, `Status` и `ReferencePerson`. Шаблон Null Object позволяет использовать пустой экземпляр объекта вместо пустого значения (термин "пустой" в данном случае означает устанавливаемые по умолчанию значения членов, например, `string.Empty` для членов типа `string`). Благодаря этому становится более удобной для чтения запись через точку, как например:

```
this.NoNulls.At.All.Here.Description
```

Что же касается базы данных, то внешние соединения можно сократить и чаще использовать внутренние соединения, поскольку внешние ключи будут, по крайней мере, указывать на пустые символы, тогда как столбцы с внешними ключами не будут пустыми. Таким образом, пустые объекты заметно упрощают дело, причем совершенно неожиданным образом.

Как следует из рис. 4.11, класс `Order` был дополнен методом `AddOrderLine()` для ввода строк заказа. Это частичный рефакторинг кода типа Encapsulate Collection (Инкапсуляция совокупности объектов) [Fowler R]. По существу, это означает, что родительский объект будет защищать все изменения в совокупности объектов.

С другой стороны, база данных — это последний сторожевой пост, и поэтому данное правило относится и к ней.

10. Сохранение заказа и его строк должно быть атомарным

В данном случае классы `Order` и `OrderLine` представляются мне как агрегат. Именно на это утверждение и будет ориентировано решение данной задачи.

Для отслеживания экземпляров измененных (новых и удаленных) объектов можно воспользоваться реализацией шаблона Unit of Work (Единица работы) [Fowler PoEAA]. Этот шаблон согласует подобные изменения, используя одну физическую операцию обращения к базе данных в состоянии сохраняемости.

11. У заказов должно быть состояние подтверждения, изменяемое пользователем

Как установлено выше, у заказов имеется состояние подтверждения (рис. 4.12), для чего достаточно ввести метод `Accept()`. А вопрос о том, следует ли использовать внутренним образом шаблон State (Состояние) [GoF Design Patterns], придется отложить до более подходящего момента. Подобные решения лучше всего принимать в ходе рефакторинга кода.

При рассмотрении остальных состояний заказа мы можем ввести дополнительные методы. А до тех пор, пока в этом нет явной необходимости, ограничимся методом `Accept()`. Этот замысел можно представить в данный момент следующим образом:

```
[Test]
public void CanAcceptOrder()
{
  Customer newCustomer = _CreateACustomer("Karlskrona");
  Order newOrder = OrderFactory.CreateOrder(newCustomer);

  Assert.IsFalse(newOrder.Status == OrderStatus.Accepted);

  newOrder.Accept();

Assert.IsTrue(newOrder.Status == OrderStatus.Accepted);
}
```

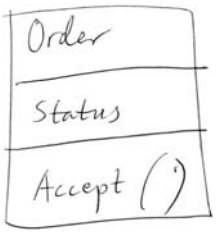

Рис. 4.12. Классы `Order` и `Status`

Модель предметной области на данном этапе

Если подвести итог всему сказанному выше, то модель предметной области можно представить так, как показано на рис. 4.13.

На заметку

Как следует из рис. 4.13, класс `ReferencePerson` оказывается в двух разных агрегатах, чего нельзя сказать об экземпляре его объектов. Это характерный пример того, как статической блок-схеме недостает выразительности. Но в то же время он ясно показывает, насколько простое и лаконичное пояснение может дать блок-схема.

Согласен, что модель предметной области выглядит на данный момент не совсем упорядоченно. В дальнейшем, при обсуждении деталей, мы постараемся привести модель в порядок, разбив ее на более логичные части.

На заметку

Модель, приведенная на рис. 4.13, а также все ее части были созданы наперед, поэтому они потребуют значительного улучшения после перехода к стадии разработки приложения.

Выше была представлена лишь базовая модель предметной области, в которой отсутствуют составляющие, связанные с инфраструктурой, в том числе упоминавшийся выше шаблон Unit of Work [Fowler PoEAA]. Разумеется, это было сделано намеренно. Мы еще вернемся к вопросам инфраструктуры далее в этой книге. А пока уделим основное внимание модели предметной области.

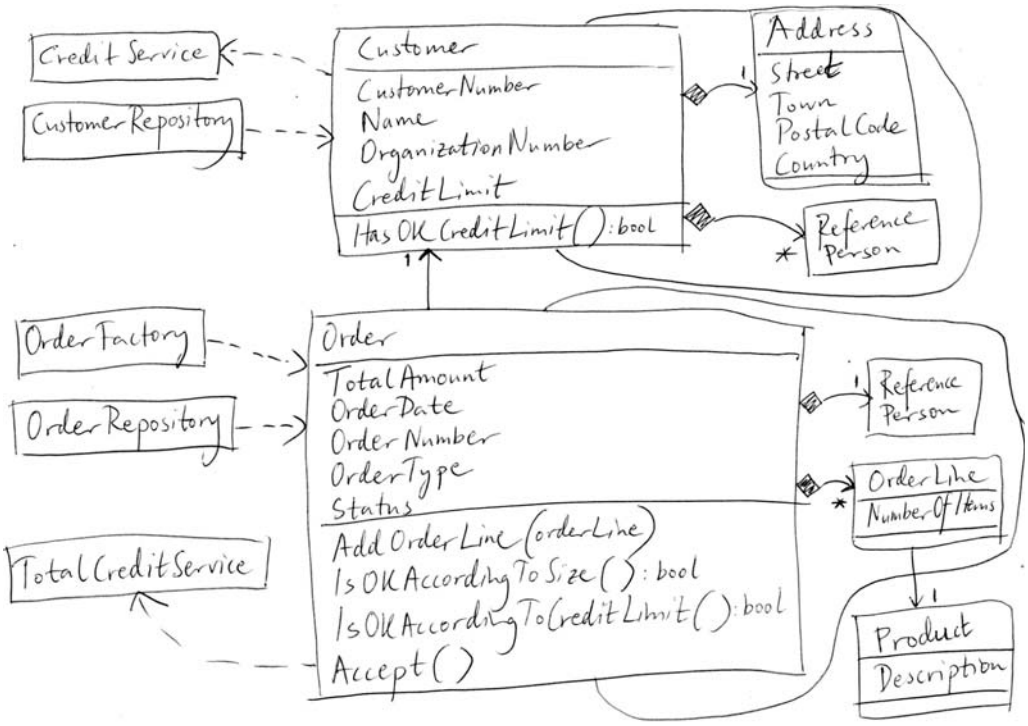

Рис. 4.13. Набросок модели предметной области, отражающий мой подход к выполнению требований по списку возможностей

Как уже отмечалось, существует множество вариантов применения шаблона Domain Model. Для того чтобы показать некоторые из них, я попросил своих коллег описать те способы применения подобных шаблонов, которым они отдают предпочтение. Эти описания вы найдете в Приложении А.

Для того чтобы дать более ясное представление о рассмотренных выше требованиях к приложению, мне бы хотелось взглянуть на них под другим углом зрения и попутно набросать несколько форм для пользовательского интерфейса, который предстоит разработать.

Первая попытка привязать пользовательский интерфейс к модели предметной области

А теперь рассмотрим новую архитектуру снизу вверх, начав с базы данных. Один из рецензентов книги сказал мне совершенно определенно, что рассмотрение архитектуры следует начинать с точки зрения разработчика пользовательского интерфейса. Если она окажется неудачной с этой точки зрения, то продолжать чтение нет смысла. У такого подхода, конечно, есть свои преимущества, поэтому бросим первый взгляд на только что намеченную модель предметной области с точки зрения разработчика пользовательского интерфейса.

Основная цель

Итак, посмотрим, насколько нам удалось добиться одной из основных целей: "упрощение слева" или же "упрощение сверху" — в зависимости от того, как представлять слои в многослойной архитектуре. Под этим подразумевается предоставление простого интерфейса API разработчику пользовательского интерфейса, чтобы ему было легче представить себе модель, а значит, уделить больше внимания вопросам пользовательского интерфейса, не думая о сложных протоколах для модели предметной области.

При первом обсуждении пользовательского интерфейса мы будем касаться привязки данных, хотя это далеко не основная цель данного обсуждения. Несмотря на то что в этой книге не уделяется много внимания привязке данных, мы все же коснемся этого вопроса в последующих главах.

Акцент на простой пользовательский интерфейс

В данный момент модель предметной области кажется слишком абстрактной, но если рассматривать ее с точки зрения пользовательского интерфейса, это положение может немного измениться. Итак, мы должны опробовать два следующих варианта:

- Составление списка заказов клиента.
- Ввод заказа.

Опять же, мы не будем касаться здесь обычной привязки данных, а лишь рассмотрим простой и ясный код для привязки пользовательского интерфейса к модели предметной области.

Составление списка заказов клиента

Итак, требуется сформировать список заказов клиента. При последующем построении модели предметной области останется только ввести новое "представление" выше этой модели и сымитировать некоторые заказы. В данном случае речь идет о такой форме, как на рис. 4.14.

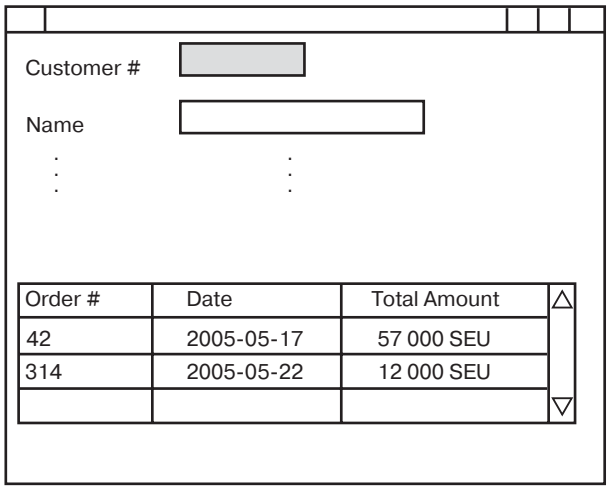

Рис. 4.14. Составление списка заказов клиента

На заметку

У вас может сложиться впечатление, что я избегаю использования таких вспомогательных средств, как редактор VS.NET. Но я решил наглядно представить общий замысел формы с помощью ручки и бумаги потому, что это лишь ранняя стадия эскизного проекта. (Иллюстрации к этой книге визуализированы мной в графической программе.)

Данная форма достаточно проста. Для ее проверки и отображения нужно написать функцию, которую можно было бы вызывать из функции `Main()` для имитации клиента и некоторых его заказов.

Пользователь, скорее всего, будет выбирать клиента из формы со списком клиентов, и поэтому мы предоставим экземпляр объекта `Customer` для конструктора формы, чтобы отобразить подробные сведения о клиенте (дадим этой форме название `CustomerForm`). Экземпляр объекта `Customer` будет храниться в частном поле формы под названием `_customer`.

В таком случае код метода `_PaintCustomer()` для вывода данных о клиенте в этой форме может выглядеть следующим образом:

```
// Форма CustomerForm
private void _PaintCustomer()
{
  // Получить данные:
  IList theOrders = _orderRepository.GetOrders(_customer);

  // Отобразить данные о клиенте:
```

```
  txtCustomerNumber.Text =
    _customer.CustomerNumber.ToString();
  txtName.Text = _customer.Name;

  // Отобразить данные о заказах:
  foreach (Order o in theOrders)
  {
  // ВЫПОЛНИТЬ...
  }
}
```

Следует подчеркнуть, что для отображения данных о заказах в приведенном выше коде можно применить шаблон Null Object [Woolf Null Object]. В этом случае у всех заказов должен быть пустой объект `ReferencePerson` — даже если ответственное лицо еще не указано. Это избавляет от необходимости проверять пустые значения, что, на мой взгляд, служит характерным примером детали, упрощающей задачу разработчика пользовательского интерфейса.

Написание кода для пользовательского интерфейса вручную оказалось несколько утомительным занятием. Но в итоге код все равно получился не очень простым, хотя были предприняты попытки сделать лишь самое элементарное.

Ввод заказа

Рассмотрим еще один пример формы, в которой можно вводить заказ. Предположим, что сначала выбирается клиент, а затем — такая форма, как на рис. 4.15.

Для этого сначала создается экземпляр объекта `Order`, после чего этот объект передается конструктору формы.

В таком случае код метода `_PaintCustomer()` может выглядеть следующим образом:

```
// Метод OrderForm._PaintOrder()
txtOrderNumber.Text = _ShowOrderNumber(_order.OrderNumber);
txtOrderDate.Text = _order.OrderDate.ToString();
// И так далее, код в этой части совершенно такой же,
// как и в приведенном выше примере кода для
// метода _PaintCustomer().
```

Для того чтобы исключить нулевое значение номера заказа (`txtOrderNumber`), в приведенном выше фрагменте кода использована вспомогательная функция (`_ShowOrderNumber()`), которая должна отображать текст типа “Новый заказ” вместо 0.

Что получается в итоге

Такой пользовательский интерфейс можно, в частности, описать с помощью шаблона Separated Presentation (Раздельное представление) [Fowler PoEAA2], чтобы отделить логику от кода, манипулирующего представлением данных. Этот основной

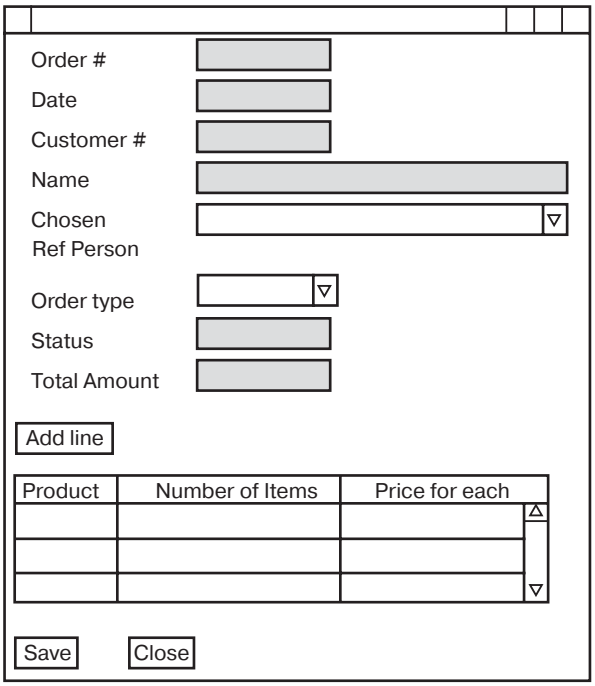

Рис. 4.15. Форма для ввода заказа

принцип рекомендуется уже давно, но зачастую им сильно злоупотребляют. Мы же воспользуемся им в какой-то мере автоматически и без всяких затрат, поскольку выбрали стиль ППО.

И в этом разделе мы попытались лучше представить себе требования к приложению, рассмотрев их с точки зрения пользовательского интерфейса. Впрочем, изучение конкретных вопросов, связанных с пользовательским интерфейсом, придется отложить вплоть до главы 11.

А теперь посмотрим на требования к приложению под еще одним углом зрения.

Еще одно измерение

До сих пор речь в этой главе шла о логической структуре модели предметной области, что, конечно, позволяет представить общую картину лишь в одном измерении. А ведь существуют и другие измерения. Рассмотрим другие стили исполнения данной модели, начав с подготовки среды развертывания по списку возможностей. В этой среде мы создаем, как упоминалось выше, клиентское приложение с расширенными функциональными возможностями (WinForms) без сервера приложений (рис. 4.16).

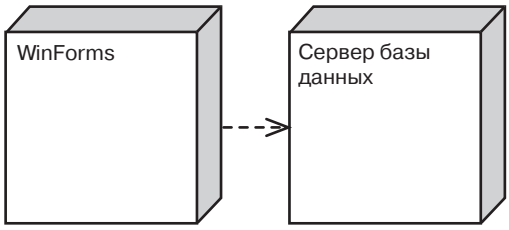

Рис. 4.16. Приложение WinForms без сервера приложений

Ради удобства изложения усложним немного ситуацию, введя требование сервера приложений, как показано на рис. 4.17.

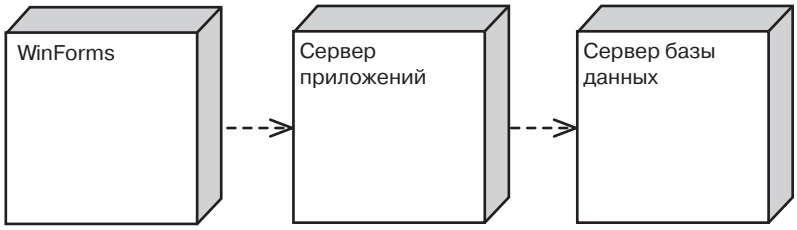

Рис. 4.17. Приложение WinForms и сервер приложений

Прежде всего, возникает вопрос: на каком уровне следует исполнять модель предметной области? Следует ли это сделать на сервере приложений, а затем передать на уровень потребителя структуры данных, используя, например, шаблон Data Transfer Objects (DTO — Объекты для передачи данных) [Fowler PoEAA]? Или же модель предметной области следует исполнять только на стороне потребителя, наполнив ее данными по запросу DTO из сервера приложений? А может быть модель предметной области следует исполнять на обоих уровнях или же должны быть две разные модели?

На заметку

Очень важно подумать о назначении сервера приложений: действительно ли он нужен и нужно ли на самом деле распределять модель предметной области? Напомним первый закон распределенного объектного проектирования: “Не распределять объекты!” [Fowler PoEAA].

Представим на мгновение, что модель предметной области следует исполнять, по крайней мере, на сервере приложений. Нужно ли тогда разделять реализацию этой модели в виде экземпляров таким образом, чтобы один экземпляр служил для представления одного клиента всем пользователям? Или же следует выделять по одному экземпляру модели предметной области на каждого пользователя либо на каждый

сеанс работы в системе, чтобы несколько экземпляров представляли конкретного клиента в определенный момент времени?

Третий вопрос: должна ли реализация модели предметной области сохранять свое состояние в промежутке между вызовами или же она должна исчезать после каждого вызова?

Четвертый вопрос: следует ли стремиться к полной реализации модели предметной области, выбирая как можно больше данных при каждом запросе, и вообще не удалять ее экземпляры?

Мы поставили немало вопросов, и ответы на них, как всегда, зависят от конкретной ситуации. Обсуждение этих вопросов усложняется, в частности, тем, что, действуя в стиле ППО, мы почти не ничего не решили в отношении инфраструктуры.

Тем не менее мне бы хотелось поделиться с вами тем, как я обычно решаю подобные вопросы.

На заметку

Под "реализацией модели предметной области" я подразумеваю создание ряда экземпляров этой модели вместо классов. Разумеется, классы разделяются среди пользователей намного чаще, чем экземпляры. Тем не менее я считаю, что этот термин вносит ясность в обсуждение.

Расположение модели предметной области

Прежде всего, если требуется контролировать уровни потребителя и сервера приложений, то было бы неплохо раскрыть и использовать модель предметной области как на уровне потребителя, так и на уровне сервера приложений. Если же она не используется на обоих уровнях, то существует риск выполнить лишнюю работу и добиться меньшей эффективности, поскольку в этом случае мы можем создать две одинаковых, хотя и не полностью, модели предметной области: одну — для потребителя, а другую — для сервера приложений. Кроме того, нам потребуются средства сопряжения для перехода от одной модели к другой. (При этом очень важно понимать саму ситуацию — с одной или двумя моделями. Если вы думаете, что модель одна, а на самом деле их две, то ситуация может осложниться.)

Следует также заметить, что в сложных ситуациях я предпочитаю использовать шаблон Presentation Model (Модель представления) [Fowler PoEAA2] для представления оптимизированного пользовательского интерфейса или варианта модели предметной области.

Изолирование или совместное использование экземпляров

Следует ли разделять реализацию модели предметной области на сервере приложений среди пользователей или сеансов работы в системе? Мне лично по душе сама идея разделяемой (т.е. совместно используемой) реализации модели предметной области, но на практике я чаще всего держусь от нее подальше. Воплотить эту идею

в жизнь намного сложнее, чем кажется на первый взгляд, поэтому я реализую модель предметной области для каждого пользователя, а зачастую и для каждого сеанса работы в системе. Ведь сделать это намного проще.

Одна из основных трудностей, связанных с разделяемым множеством экземпляров объектов предметной области, возникает в том случае, если требуется исполнение в распределенном виде — возможно, на двух серверах приложений в "ничего не разделяющем" кластере. Это, в свою очередь, приводит к довольно сложной проблеме распределенного кэширования — по крайней мере, в отношении обеспечения согласованности в реальном масштабе времени. Данная проблема настолько сложна, что я пока что отказываюсь ее решать и стараюсь найти более общее решение, которое может оказаться вполне пригодным для вертикального масштабирования системы. Но в особых случаях все же можно прийти к вполне приемлемому решению. Конечно, если все экземпляры модели предметной области находятся в оперативной памяти, то лишь в немногих ситуациях может потребоваться горизонтальное масштабирование системы — по крайней мере, по соображениям эффективности. Тем не менее, если возникает столь сложная проблема, найти ее удачное решение оказывается непросто.

На заметку

Здесь необходимо вкратце пояснить, что означает "ничего не разделяющий" кластер. Этот термин означает, что серверы приложений не используют совместно ЦП, диск или оперативную память. Они действуют совершенно независимо друг от друга, что выгодно с точки зрения масштабируемости.

Подробнее об этом можно прочитать в книге *In Search of Clusters* [Pfister Wolfpack].

Реализация модели предметной области с сохранением или без сохранения состояния

На сервере приложений я не допускаю сохранение состояния реализации модели предметной области и избавляюсь от нее в промежутке между вызовами. А на стороне потребителя я стараюсь сохранить реализацию модели предметной области в промежутке между вызовами, но зачастую делаю это только после прецедента использования.

Полная или частичная реализация модели предметной области

И наконец, я не стремлюсь реализовать модель предметной области полностью. Делать это на стороне потребителя, где базы данных, как правило, достаточно крупные, было бы просто неразумно. Вместо этого я придерживаюсь старой истины: брать только то, что нужно, и ничего лишнего. После того как модель будет реализована, я избавляюсь от нее, освобождая место для других данных, и не храню старые данные слишком долго. Но совсем другое дело — статические данные. Мы должны как можно больше кэшировать данные "только для чтения".

На заметку

Полные, разделяемые, сохраняющие свое состояние реализации модели предметной области поддерживаются в одном открытом проекте под названием Prevayler [Prevayler], а также в других аналогичных проектах. Это означает, что модель предметной области и базу данных можно в какой-то степени рассматривать как одно и то же. А для того чтобы это стало возможным, нам потребуются крупные объемы оперативной памяти, хотя в настоящее время это не проблема, поскольку цены на оперативную память постоянно падают, а 64-разрядные машины стали применяться повсеместно.

Основная идея состоит в том, что при каждом изменении в базе данных осуществляется соответствующая запись в журнале последовательной регистрации. При отсутствии электроэнергии в сети реализацию модели предметной области можно восстановить, прочитав ее копию, сохраненную в определенный момент времени, а также содержимое журнала регистрации. Это довольно простой и весьма эффективный стиль исполнения модели, поскольку все данные находятся в оперативной памяти.

По существу, мы имеем дело с объектной, но простой базой данных, поскольку отпадает необходимость выявлять и устранять неисправности и выполнять прочие типичные процедуры технического обслуживания базы данных. Все экземпляры постоянно находятся в оперативной памяти, и поэтому не требуется объектно-реляционное преобразование как в направлении реляционной (и даже объектной) базы данных, так и обратно. Остается только реализация модели предметной области.

Мы не станем усложнять ситуацию далее в этой книге из-за сервера приложений и будем считать, что мы работаем с клиентским приложением, имеющим расширенные функциональные возможности, включая модель предметной области или Web-приложение, т.е. речь опять-таки пойдет непосредственно о модели предметной области.

Такое решение объясняется, в частности, тем, что классический метод использования серверов приложений постепенно уходит в прошлое или, по крайней мере, меняется. А что же теперь актуально? Возможно, архитектура, ориентированная на службы (АОС), но это уже совсем другая тема. Впрочем, мы слегка затронем ее в главе 10.

Резюме

Эту главу можно было бы продолжать до бесконечности, но, к сожалению, ее придется заканчивать. В ней были рассмотрены вопросы, касающиеся “новой” используемой по умолчанию архитектуры и моделей предметной области, а также было намечено в общих чертах ее применение на одном примере приложения. А теперь настало время для более углубленного исследования модели предметной области и внесения многочисленных изменений в первоначальный набросок проекта, сделанный в этой главе.