UML Distilled

A Brief Guide to the Standard Object Modeling Language

Third Edition

Martin Fowler

Addison-Wesley

3

Диаграммы классов: основы

Если кто-нибудь подойдет к вам в темном переулке и спросит: «Хотите посмотреть на диаграмму UML?», знайте – скорее всего, речь идет о диаграмме класса. Большинство диаграмм UML, которые я встречал, были диаграммами классов. Помимо своего широкого применения диаграммы классов концентрируют в себе большой диапазон понятий моделирования. Хотя их основные элементы используются практически всеми, более сложные элементы применяются не так часто. Именно поэтому я разделил рассмотрение диаграмм классов на две части: основы (данная глава) и дополнительные понятия (глава 5).

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами. В UML термин **функциональность** (feature) применяется в качестве основного термина, описывающего и свойства, и операции класса.

На рис. 3.1 изображена типичная модель класса, понятная каждому, кто имел дело с обработкой заказов клиентов. Прямоугольники на диаграмме представляют классы и разделены на три части: имя класса (жирный шрифт), его атрибуты и его операции. На рис. 3.1 также показаны два вида связей между классами: ассоциации и обобщения.

Свойства

Свойства представляют структурную функциональность класса. В первом приближении можно рассматривать свойства как поля класса. Как мы увидим позднее, в действительности это не так просто, но вполне приемлемо для начала.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях. Хотя на диаграмме они выглядят совершенно по-разному, в действительности это одно и то же.

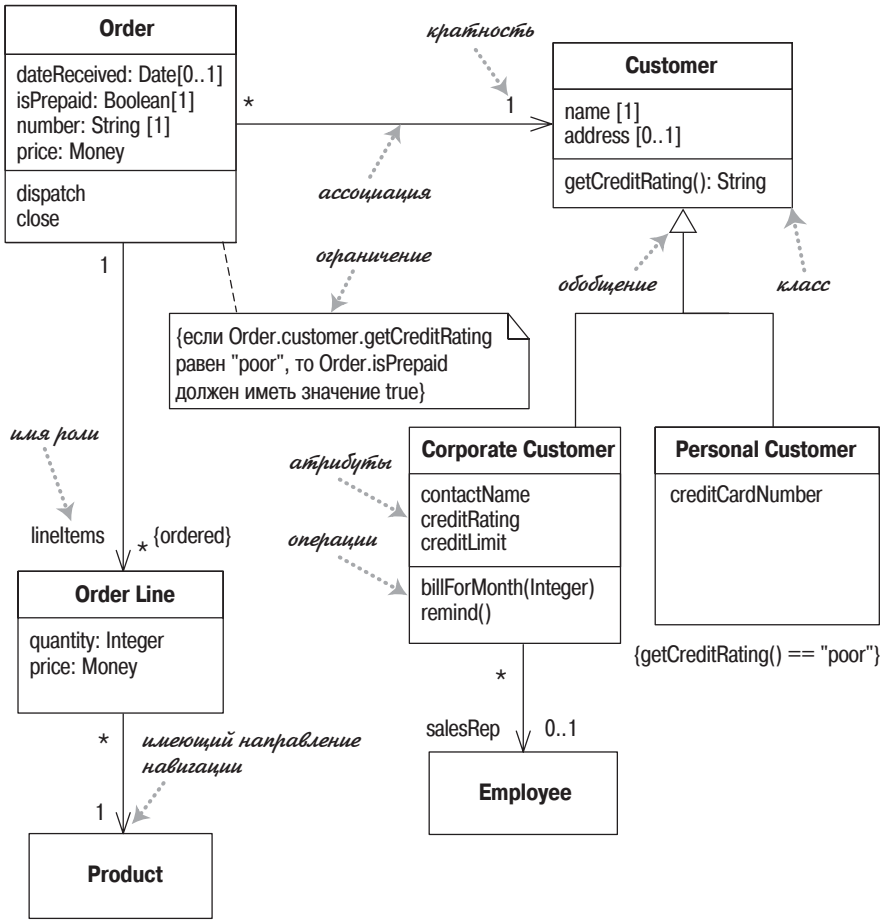

Рис. 3.1. *Простая диаграмма класса*

Атрибуты

Атрибут описывает свойство в виде строки текста внутри прямоугольника класса. Полная форма атрибута:

```
видимость имя: тип кратность = значение по умолчанию {строка свойств}
```

Например:

```
- имя: String [1] = "Без имени" {readOnly}
```

Обязательно только имя.

- Метка `видимость` обозначает, относится ли атрибут к открытым (+) (public) или к закрытым (-) (private). Другие типы видимости обсуждаются на *стр. 110*.

- Имя атрибута – способ ссылки класса на атрибут – приблизительно соответствует имени поля в языке программирования.
- Тип атрибута накладывает ограничение на вид объекта, который может быть размещен в атрибуте. Можно считать его аналогом типа поля в языке программирования.
- Кратность рассмотрена на *стр. 65*.
- Значение по умолчанию представляет собой значение для вновь создаваемых объектов, если атрибут не определен в процессе создания.
- Элемент {строка свойств} позволяет указывать дополнительные свойства атрибута. В примере он равен {readOnly}, то есть клиенты не могут изменять атрибут. Если он пропущен, то, как правило, атрибут можно модифицировать. Остальные строки свойств будут описаны позже.

Ассоциации

Другая ипостась свойства – это ассоциация. Значительная часть информации, которую можно указать в атрибуте, появляется в ассоциации. На рис. 3.2 и 3.3 показаны одни и те же свойства, представленные в различных обозначениях.

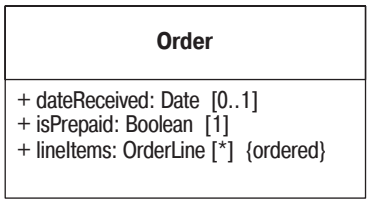

Рис. 3.2. *Представление свойств заказа в виде атрибутов*

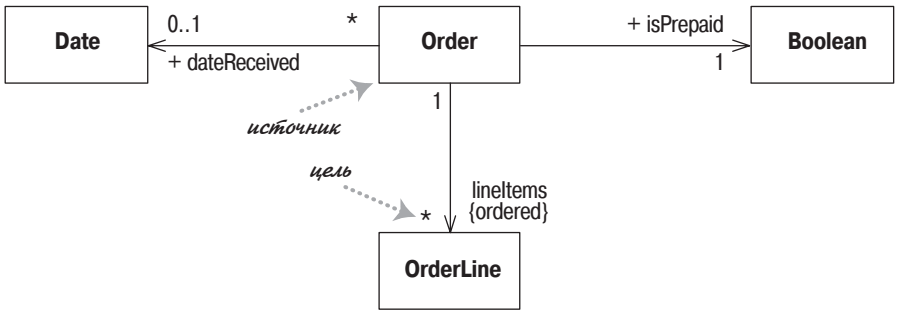

Рис. 3.3. *Представление свойств заказа в виде ассоциаций*

Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации. Целевой

конец ассоциации указывает на класс, который является типом свойства. Большая часть информации в обоих представлениях одинакова, но некоторые элементы отличаются друг от друга. В частности, ассоциация может показывать кратность на обоих концах линии.

Естественно, возникает вопрос: «Когда следует выбирать то или иное представление?». Как правило, я стараюсь обозначать при помощи атрибутов небольшие элементы, такие как даты или логические значения, – главным образом, типы значений (*стр. 101*), – а ассоциации для более значимых классов, таких как клиенты или заказы. Я также предпочитаю использовать прямоугольники классов для наиболее значимых классов диаграммы, а ассоциации и атрибуты для менее важных элементов этой диаграммы.

Кратность

Кратность свойства обозначает количество объектов, которые могут заполнять данное свойство. Чаще всего встречаются следующие кратности:

- 1 (Заказ может представить только один клиент.)
- 0..1 (Корпоративный клиент может иметь, а может и не иметь единственного торгового представителя.)
- * (Клиент не обязан размещать заказ, и количество заказов не ограничено. Он может разместить ноль или более заказов.)

В большинстве случаев кратности определяются своими нижней и верхней границами, например 2..4 для игроков в канасту. Нижняя граница может быть нулем или положительным числом, верхняя граница представляет собой положительное число или * (без ограничений). Если нижняя и верхняя границы совпадают, то можно указать одно число; поэтому 1 эквивалентно 1..1. Поскольку это общий случай, * является сокращением 0..*.

При рассмотрении атрибутов вам могут встретиться термины, имеющие отношение к кратности.

- **Optional** (необязательный) предполагает нулевую нижнюю границу.
- **Mandatory** (обязательный) подразумевает, что нижняя граница равна или больше 1.
- **Single-valued** (однозначный) – для такого атрибута верхняя граница равна 1.
- **Multivalued** (многозначный) имеет в виду, что верхняя граница больше 1; обычно *.

Если свойство может иметь несколько значений, я предпочитаю употреблять множественную форму его имени. По умолчанию элементы с множественной кратностью образуют множество, поэтому если вы просите клиента разместить заказы, то они приходят не в произволь-

ном порядке. Если порядок заказов в ассоциации имеет значение, то в конце ассоциации необходимо добавить {ordered}. Если вы хотите разрешить повторы, то добавьте {nonunique}. (Если желательно явным образом показать значение по умолчанию, то можно использовать {unordered} и {unique}.) Встречаются также имена для unordered, nonunique, ориентированные на коллекции, такие как {bag}.

UML 1 допускал дискретные кратности, например 2,4 (означающую 2 или 4, как в случае автомобилей, до того как появились минивэны). Дискретные кратности не были широко распространены, и в UML 2 их уже нет.

Кратность атрибута по умолчанию равна [1]. Хотя это и верно для метамодели, нельзя предполагать, что если значение кратности для атрибута на диаграмме опущено, то оно равно [1], поскольку информация о кратности на диаграмме может отсутствовать. Поэтому я предпочитаю указывать кратность явным образом, если эта информация важна.

Программная интерпретация свойств

Как и для других элементов UML, интерпретировать свойства в программе можно по-разному. Наиболее распространенным представлением является поле или свойство языка программирования. Так, класс Order Line (Строка заказа), показанный на рис. 3.1, мог бы быть представлен в Java следующим образом:

```
public class OrderLine...
  private int quantity;
  private Money price;
  private Order order;
  private Product product
```

В языке, подобном C#, который допускает свойства, это могло бы выглядеть так:

```
public class OrderLine ...
  public int Quantity;
  public Money Price;
  public Order Order;
  public Product Product;
```

Обратите внимание, что атрибут обычно соответствует открытым (public) свойствам в языке, поддерживающем свойства, но соответствует закрытым (private) полям в языке, в котором такой поддержки нет. В языке без свойств с полями можно общаться посредством методов доступа (получение и установка). У атрибута только для чтения не будет метода установки (в случае полей) или операции установки (в случае свойства). Учтите, что если свойству не присвоить имя, то в общем случае ему будет назначено имя целевого класса.

Применение закрытых полей является интерпретацией, ориентированной сугубо на реализацию. Интерпретация, ориентированная в большей степени на интерфейс, может быть акцентирована на методах доступа, а не на данных. В этом случае атрибуты класса `Order Line` могли бы быть представлены следующими методами:

```
public class OrderLine...
  private int quantity;
  private Product product;
  public int getQuantity() {
    return quantity;
  }
  public void setQuantity(int quantity) {
    this.quantity = quantity;
  }
  public Money getPrice() {
    return product.getPrice().multiply(quantity);
  }
```

Здесь нет поля для цены – ее значение вычисляется. Но поскольку клиенты класса `Order Line` заинтересованы в этой информации, она выглядит как поле. Клиенты не могут сказать, что является полем, а что вычисляется. Такое сокрытие информации представляет сущность инкапсуляции.

Если атрибут имеет несколько значений, то связанные с ним данные представляют собой коллекцию. Поэтому класс `Order` (Заказ) будет ссылаться на коллекцию классов `Order Line`. Поскольку эта кратность упорядочена (ordered), то и коллекция должна быть упорядочена (например, List в Java или IList в .NET). Если коллекция не упорядочена, то, строго говоря, она не должна иметь ярко выраженного порядка, то есть должна быть представлена множеством, но большинство специалистов реализуют неупорядоченные атрибуты также в виде списков. Некоторые разработчики используют массивы, но поскольку UML подразумевает неограниченность сверху, то я почти всегда для структуры данных применяю коллекцию.

Многозначные свойства имеют интерфейс, отличный от интерфейса свойств с одним значением (в Java):

```
class Order {
  private Set lineItems = new HashSet();
  public Set getLineItems() {
    return Collections.unmodifiableSet(lineItems);
  }
  public void addLineItem (OrderItem arg) {
    lineItems.add (arg);
  }
  public void removeLineItem (OrderItem arg) {
    lineItems.remove(arg);
  }
```

В большинстве случаев значения многозначных свойств не присваиваются прямо; вместо этого применяются методы добавления (add) и удаления (remove). Для того чтобы управлять своим свойством `Line Items` (Позиции заказов), заказ должен контролировать членство этой коллекции; поэтому он не должен передавать незащищенную коллекцию. В таких случаях я использовал представителя защиты, чтобы заключить коллекцию в оболочку только для чтения. Можно также реализовать необновляемый итератор или сделать копию. Конечно, так клиентам удобнее модифицировать объекты-члены, но они не должны иметь возможность напрямую изменять саму коллекцию.

Поскольку многозначные атрибуты подразумевают коллекции, то практически вы никогда не увидите классы коллекций на диаграмме класса. Их можно увидеть только на очень низком уровне представления диаграмм самих коллекций.

Необходимо крайне остерегаться классов, являющихся не чем иным, как коллекциями полей и средствами доступа к ним. Объектно-ориентированное проектирование должно предоставлять объекты с богатым поведением, поэтому они не должны просто обеспечивать данными другие объекты. Если данные запрашиваются многократно с помощью средств доступа, то это сигнал к тому, что такое поведение должно быть перенесено в объект, владеющий этими данными.

Эти примеры также подтверждают тот факт, что между UML и программой нет обязательного соответствия, однако есть подобие. Соглашения, принятые внутри команды разработчиков, приведут к более полному соответствию.

Независимо от того, как реализовано свойство – в виде поля или как вычисляемое значение, оно представляет нечто, что объект может всегда предоставить. Не следует прибегать к свойству для моделирования транзитного отношения, такого, когда объект передается в качестве параметра во время вызова метода и используется только в рамках данного взаимодействия.

Двунаправленные ассоциации

До сих пор мы говорили об однонаправленных ассоциациях. К другому распространенному типу ассоциаций относится двунаправленная ассоциация, например, показанная на рис. 3.4.

Двунаправленная ассоциация – это пара свойств, связанных в противоположных направлениях. Класс `Car` (Автомобиль) имеет свойство

Рис. 3.4. *Двунаправленная ассоциация*

owner:Person[1], а класс Person (Личность) имеет свойство cars:Car[*]. (Обратите внимание, что я использовал множественную форму имени свойства cars, а это соглашение общепринятое, но ненормативное.)

Обратная связь между ними подразумевает, что если вы следуете обоим свойствам, то должны вернуться обратно к множеству, содержащему вашу исходную точку. Например, если я начинаю с конкретной модели MG Midget, нахожу ее владельца, а затем смотрю на множество принадлежащих ему машин, то оно должно включать модель Midget, с которой я начал.

В качестве альтернативы маркировки ассоциации по свойству многие люди, особенно если они имеют опыт моделирования данных, любят именовать ассоциации с помощью глаголов (рис. 3.5), чтобы отношение можно было использовать в предложении. Это вполне допустимо, и можно добавить к ассоциации стрелку, чтобы избежать неопределенности. Большинство разработчиков объектов предпочитают использовать имя свойства, так как оно больше соответствует функциональным назначениям и операциям.

Некоторые разработчики тем или иным способом именуют каждую ассоциацию. Я предпочитаю давать имя ассоциации, только если это улучшает понимание. Слишком часто встречаются такие имена, как «has» (имеет) или «is related to» (связан с).

На рис. 3.4 двунаправленная природа ассоциации подчеркивается стрелками на обоих концах ассоциации. На рис. 3.5 стрелок нет; в языке UML эта форма применяется либо для обозначения двунаправленной ассоциации, либо когда направление отношения не показывается. Я предпочитаю обозначать двунаправленную ассоциацию с помощью двойных стрелок.

Рис. 3.5. *Использование глагола (own – владеть) в имени ассоциации*

Реализация двунаправленной ассоциации в языке программирования часто представляет некоторую сложность, поскольку необходимо обеспечить синхронизацию обоих свойств. В C# для реализации двунаправленной ассоциации я делаю следующее:

```
class Car...
  public Person Owner {
    get {return__owner;}
    set {
      if (_owner != null) __owner.friendCars().Remove(this);
      __owner = value;
      if (_owner != null)__owner.friendCars().Add(this);
```

```
      }
    }
    private Person _owner;
...
class Person ...
    public IList Cars {
      get {return ArrayList.ReadOnly(_cars);}
    }
    public void AddCar(Car arg) {
      arg.Owner = this;
    }
    private IList _cars = new ArrayList();
    internal IList friendCars() {
      //должен быть использован только Car.Owner
      return _cars;
    }
....
```

Главное – сделать так, чтобы одна сторона ассоциации (по возможности с единственным значением) управляла всем отношением. Для этого ведомый конец (Person) должен предоставить инкапсуляцию своих данных ведущему концу. Это приводит к добавлению в ведомый класс не очень удобного метода, которого здесь не должно было бы быть в действительности, если только язык не имеет более тонкого инструмента управления доступом. Я здесь употребил слово «friend» (друг) в имени как намек на C++, где метод установки ведущего класса действительно был бы дружественным. Как и большинство кода, работающего со свойствами, это стереотипный фрагмент, и поэтому многие разработчики предпочитают получать его посредством различных способов генерации кода.

В концептуальных моделях навигация не очень важна, поэтому в таких случаях я не показываю каких-либо навигационных стрелок.

Операции

Операции (operations) представляют собой действия, реализуемые некоторым классом. Существует очевидное соответствие между операциями и методами класса. Обычно можно не показывать такие операции, которые просто манипулируют свойствами, поскольку они и так подразумеваются.

Полный синтаксис операций в языке UML выглядит следующим образом:

```
видимость имя (список параметров) : возвращаемый тип {строка свойств}
```

- Метка видимости обозначает, относится ли операция к открытым (+) (public) или к закрытым (-) (private); другие типы видимости обсуждаются на *стр. 110*.

- `Имя` – это строка.
- `Список параметров` – список параметров операции.
- `Возвращаемый тип` – тип возвращаемого значения, если таковое есть.
- `Строка свойств` – значения свойств, которые применяются к данной операции.

Параметры в списке параметров обозначаются таким же образом, что и для атрибутов. Они имеют вид:

```
направление имя: тип = значение по умолчанию
```

- Имя, тип и значение по умолчанию те же самые, что и для атрибутов.
- Направление обозначает, является ли параметр входным (`in`), выходным (`out`) или тем и другим (`inout`). Если направление не указано, то предполагается `in`.

Например, в счете операция может выглядеть так:

```
+ balanceOn (date: Date) : Money
```

В рамках концептуальной модели не следует применять операции для спецификации интерфейса класса. Вместо этого используйте их для представления главных обязанностей класса, возможно, с помощью пары слов, обобщающих ответственность в CRC-карточках (*стр. 89*).

По моему мнению, следует различать операции, изменяющие состояние системы, и операции, не делающие этого. Язык UML определяет **запрос** как некую операцию, результатом которой является некоторое значение, получаемое от класса; при этом состояние системы не изменяется, то есть данная операция не вызывает побочных эффектов. Такую операцию можно пометить строкой свойств `{query}` (запрос). Операции, изменяющие состояние, я называю **модификаторами**, иначе именуемые командами.

Строго говоря, различие между запросом и модификаторами состоит в том, могут ли они изменять видимое состояние [33]. Видимое состояние – это то, что можно наблюдать извне. Операция, обновляющая кэш, изменит внутреннее состояние, но это не окажет никакого влияния на то, что видно снаружи.

Я считаю полезным выделение запросов, так как это позволяет изменить порядок выполнения запросов и не изменить при этом поведение системы. Общепринято конструировать операции так, чтобы модификаторы не возвращали значение, – тогда можно быть уверенным в том, что операции, возвращающие значения, являются запросами. [33] называет это принципом разделения команды-запроса. Делать так все время не очень удобно, но необходимо применять этот способ так часто, как только возможно.

Другие термины, с которыми иногда приходится сталкиваться, – это методы получения значения (getting methods) и методы установки

значения (setting methods). **Метод получения значения** возвращает некоторое значение из поля (и не делает ничего больше). **Метод установки значения** помещает некоторое значение в поле (и не делает ничего больше). За пределами класса клиент не способен определить, является ли запрос методом получения значения или модификатор – методом установки значений. Эта информация о методах является исключительно внутренней для каждого из классов.

Существует еще различие между операцией и методом. **Операция** представляет собой то, что вызывается объектом – объявление процедуры, тогда как **метод** – это тело процедуры. Эти два понятия различают, когда имеют дело с полиморфизмом. Если у вас есть супертип с тремя подтипами, каждый из которых переопределяет одну и ту же операцию супертипа, то вы имеете дело с одной операцией и четырьмя реализующими ее методами.

Обычно термины *операция* и *метод* употребляются как взаимозаменяемые, однако иногда полезно их различать.

Обобщение

Типичный пример **обобщения** (generalization) включает индивидуального и корпоративного клиентов некоторой бизнес-системы. Несмотря на определенные различия, у них много общего. Одинаковые свойства можно поместить в базовый класс `Customer` (Клиент, супертип), при этом класс `Personal Customer` (Индивидуальный клиент) и класс `Corporate Customer` (Корпоративный клиент) будут выступать как подтипы.

Этот факт служит объектом разнообразных интерпретаций в моделях различных уровней. На концептуальном уровне мы можем утверждать, что Корпоративный клиент представляет собой подтип Клиента, если все экземпляры класса Корпоративный клиент по определению являются также экземплярами класса Клиент. Таким образом, класс Корпоративный клиент представляет собой частную разновидность класса Клиент. Основная идея заключается в следующем: все, что нам известно о классе Клиент (ассоциации, атрибуты, операции), справедливо также и для класса Корпоративный клиент.

С точки зрения программного обеспечения очевидная интерпретация наследования выглядит следующим образом: Корпоративный клиент является подклассом класса Клиент. В основных объектно-ориентированных языках подкласс наследует всю функциональность суперкласса и может переопределять любые методы суперкласса.

Важным принципом эффективного использования наследования является **замещаемость**. Мне необходимо иметь возможность подставить Корпоративного клиента в любом месте программы, где требуется Клиент, и при этом все должно прекрасно работать. По существу это означает, что когда я пишу программу в предположении, что у меня есть Клиент, то я могу свободно использовать любой подтип Клиента.

Вследствие полиморфизма Корпоративный клиент может реагировать на определенные команды не так, как другой Клиент, но вызывающий не должен беспокоиться об этом отличии. (Дополнительную информацию можно найти в главе «Liskov Substitution Principle (LSP)» (Принцип замещения Лисков) книги [30]).

Наследование представляет собой мощный механизм, но оно несет с собой много такого, что не всегда является необходимым для достижения замещаемости. Вот хороший пример: на заре существования языка Java многим разработчикам не нравилась реализация встроенного класса `Vector` (Вектор), и они хотели заменить его чем-нибудь полегче. Однако единственным способом получения класса, способного заменить `Vector`, было создание его подкласса, что означало наследование множества нежелательных данных и поведения.

Замещаемые классы можно создавать при помощи массы других механизмов. Поэтому многие разработчики предпочитают различать создание подтипа, то есть наследование интерфейса, и создание подкласса, или наследование реализации. Класс – это **подтип**, если он может замещать свой супертип, в независимости от того, использует он наследование или нет. Создание **подкласса** используется как синоним обычного наследования.

Существует достаточное количество других механизмов, позволяющих создавать подтипы без создания подклассов. Примером может служить реализация интерфейса (*стр. 96*) и множество стандартных шаблонов разработки [21].

Примечания и комментарии

Примечания – это комментарии на диаграммах. Примечания могут существовать сами по себе или быть связаны пунктирной линией с элементами, которые они комментируют (рис. 3.6). Они могут присутствовать на диаграммах любого типа.

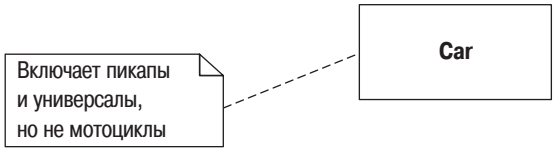

Рис. 3.6. *Примечание используется как комментарий к одному или более элементам диаграммы*

Иногда применять пунктирную линию неудобно из-за невозможности точного позиционирования конца линии. Поэтому по общепринятому соглашению в конце линии размещается небольшая открытая окружность. В некоторых случаях удобнее поместить однострочный комментарий на элементе диаграммы, при этом в начале текста ставятся два дефиса: --.

Зависимость

Считается, что между двумя элементами существует **зависимость** (dependency), если изменения в определении одного элемента **(сервера)** могут вызвать изменения в другом элементе **(клиенте)**. В случае классов зависимости появляются по разным причинам: один класс посылает сообщение другому классу; один класс владеет другим классом как частью своих данных; один класс использует другой класс в качестве параметра операции. Если класс изменяет свой интерфейс, то сообщения, посылаемые этому классу, могут стать недействительными.

По мере роста систем необходимо все более и более беспокоиться об управлении зависимостями. Если зависимости выходят из-под контроля, то каждое изменение в системе оказывает действие, нарастающее волнообразно по мере увеличения количества изменений. Чем больше волна, тем труднее что-нибудь изменить.

UML позволяет изобразить зависимости между элементами всех типов. Зависимости можно использовать всякий раз, когда надо показать, как изменения в одном элементе могут повлиять на другие элементы.

На рис. 3.7 показаны зависимости, которые можно обнаружить в многоуровневом приложении. Класс `Benefits Window` (Окно льгот) – это пользовательский интерфейс, или класс **представления**, зависящий от класса `Employee` (Сотрудник). Класс `Employee` – это **объект предметной области**, который представляет основное поведение системы, в данном случае бизнес-правила. Это означает, что если класс `Employee` изменяет свой интерфейс, то, возможно, и класс `Benefits Window` также должен измениться.

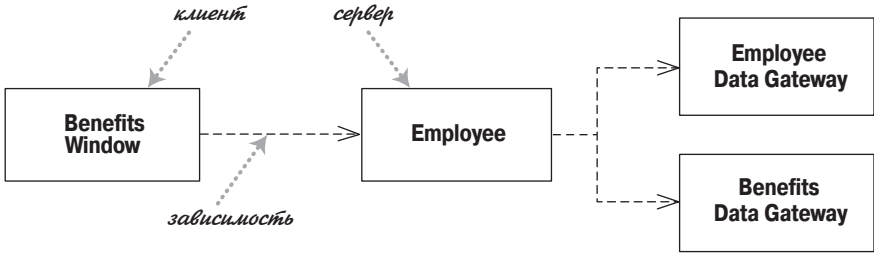

Рис. 3.7. *Пример зависимостей*

Здесь важно то, что зависимость имеет только одно направление и идет от класса представления к классу предметной области. Таким образом, мы знаем, что имеем возможность свободно изменять класс `Benefits Window`, не оказывая влияния на объект `Employee` или другие объекты предметной области. Я понял, что строгое разделение логики представления и логики предметной области, когда представление зависит от предметной области, но не наоборот, – это ценное правило, которому я должен следовать.

Второй существенный момент этой диаграммы: здесь нет прямой зависимости двух классов Data Gateway (Шлюз данных) от Benefits Window. Если эти классы изменяются, то, возможно, должен измениться и класс Employee. Но если изменяется только реализация класса Employee, а не его интерфейс, то на этом изменения и заканчиваются.

UML включает множество видов зависимостей, каждая с определенной семантикой и ключевыми словами. Базовая зависимость, которую я здесь обрисовал, с моей точки зрения, наиболее полезна, и обычно я использую ее без ключевых слов. Чтобы сделать ее более детальной, вы можете добавить соответствующее ключевое слово (табл. 3.1).

Таблица 3.1. Избранные ключевые слова зависимостей

Ключевое слово	Значение
«call» (вызывать)	Источник вызывает операцию в цели
«create» (создавать)	Источник создает экземпляр цели
«derive» (производить)	Источник представляет собой производное цели
«instantiate» (создать экземпляр)	Источник является экземпляром цели. (Обратите внимание, что если источник является классом, то сам класс является экземпляром класса класс; то есть целевой класс – это метакласс)
«permit» (разрешать)	Цель разрешает источнику доступ к ее закрытой функциональности
«realize» (реализовать)	Источник является реализацией спецификации или интерфейса, определенного целью (*стр. 96*)
«refine» (уточнить)	Уточнение означает отношение между различными семантическими уровнями; например, источник может быть классом разработки, а цель – соответствующим классом анализа
«substitute» (заместить)	Источник может быть заменен целью (*стр. 72*)
«trace» (проследить)	Используется, чтобы отследить такие моменты, как требования к классам или как изменения одной ссылки модели влияют на все остальное
«use» (использовать)	Для реализации источника требуется цель

Базовая зависимость не является транзитивным отношением. Примером **транзитивного** отношения может служить отношение «эта борода больше». Если у Джима борода больше, чем у Гради, а борода Гради больше бороды Айвара, то мы можем сделать вывод, что у Джима борода больше, чем у Айвара. Некоторые типы зависимостей, такие как замещение, являются транзитивными, но в большинстве случаев существует значительное расхождение между прямыми и обратными зависимостями, как показано на рис. 3.7.

Многие отношения UML предполагают зависимость. Направленная ассоциация от Order к Customer означает, что Order зависит от Customer. Подкласс зависит от своего суперкласса, но не наоборот.

Вашим основным правилом должна стать минимизация зависимостей, особенно когда они затрагивают значительную часть системы. В частности, будьте осторожны с циклами, поскольку они могут привести к циклическим изменениям. Я не слишком строго придерживаюсь этого правила. Я не имею в виду взаимные зависимости между тесно связанными классами, но стараюсь избегать циклов на более высоких уровнях, особенно между пакетами.

Бесполезно пытаться показать все зависимости на диаграмме классов; их слишком много, и они слишком сильно отличаются. Соблюдайте меру и показывайте только зависимости, относящиеся к конкретной теме, о которой вы хотите сообщить. Чтобы понимать и управлять зависимостями, лучше всего использовать для этого диаграммы пакетов (*стр. 114*).

В самом общем случае я использую зависимости с классами для иллюстрации транзитивного отношения, такого, когда один объект передается другому в качестве параметра. Иногда их применяют с ключевыми словами «parameter» (параметр), «local» (локальный) и «global» (глобальный). Эти ключевые слова можно увидеть на ассоциациях в моделях UML 1, и в этом случае они обозначают транзитные связи, а не свойства. Эти ключевые слова не входят в UML 2.

Зависимости можно обнаружить, просматривая программу, вот почему эти инструменты идеальны для анализа зависимостей. Применение инструментария для обращения схем зависимостей – наиболее полезный способ применения этого раздела UML.

Правила ограничений

При построении диаграмм классов большая часть времени уходит на представление различных ограничений. На рис. 3.1 показано, что Заказ (Order) может быть сделан только одним единственным Клиентом (Customer). Из этой диаграммы классов также следует, что каждая Line Item (Позиция заказа) рассматривается отдельно: вы можете заказать 40 коричневых, 40 голубых и 40 красных штучек, но не 120 штук вообще. Далее диаграмма утверждает, что Корпоративные клиенты располагают кредитами, а Индивидуальные клиенты – нет.

С помощью базовых конструкций ассоциации, атрибута и обобщения можно сделать многое, специфицируя наиболее важные ограничения, но этими средствами невозможно записать каждое ограничение. Эти ограничения еще нужно каким-то образом отобразить, и диаграмма классов является вполне подходящим местом для этого.

Язык UML разрешает использовать для описания ограничений все что угодно. При этом необходимо лишь придерживаться правила: ограничения следует помещать в фигурные скобки ({}). Можно употреблять разговорный язык, язык программирования или формальный объектный язык ограничений UML (Object Constraint Language, OCL) [43], базирующийся на исчислении предикатов. Формальное написание позволяет избежать риска неоднозначного толкования конструкций разговорного языка. Однако это приводит к возможности недоразумений из-за непрофессионального владения OCL пишущими и читающими. Поэтому до тех пор, пока ваши читатели не вполне овладеют исчислением предикатов, я предлагаю говорить на обычном языке.

Если хотите, можете предварять ограничение именем с двоеточием, например: {запрещение кровосмешения: муж и жена не должны быть родными братом и сестрой}.

Когда применяются диаграммы классов

Диаграммы классов составляют фундамент UML, и поэтому их постоянное применение является условием достижения успеха. Эта глава посвящена основным понятиям, а многие более сложные материи обсуждаются в главе 5.

Трудность, связанная с диаграммами классов, заключается в том, что они настолько обширны, что их применение может оказаться непомерно сложным. Приведем несколько полезных советов.

- Не пытайтесь задействовать сразу все доступные понятия. Начните с самых простых, описанных в этой главе: классов, ассоциаций, атрибутов, обобщений и ограничений. Обращайтесь к дополнительным понятиям, рассмотренным в главе 5, только если они действительно необходимы.
- Я пришел к выводу, что концептуальные диаграммы классов очень полезны при изучении делового языка. Чтобы при этом все получалось, необходимо всячески избегать обсуждения программного обеспечения и применять очень простые обозначения.
- Не надо строить модели для всего на свете, вместо этого следует сконцентрироваться на ключевых аспектах. Лучше создать мало диаграмм, которые постоянно применяются в работе и отражают все внесенные изменения, чем иметь дело с большим количеством забытых и устаревших моделей.

Самая большая опасность, связанная с диаграммами классов, заключается в том, что вы можете сосредоточиться исключительно на структуре и забыть о поведении. Поэтому, рисуя диаграммы классов для того, чтобы разобраться в программном обеспечении, используйте какие-либо формы анализа поведения. Если вы применяете эти методы поочередно, значит, вы двигаетесь в верном направлении.

Проектирование по контракту

Проектирование по контракту (Design by Contract) – это метод проектирования, являющийся центральным свойством языка Eiffel. И метод, и язык разработаны Бертраном Мейером [33]. Однако проектирование по контракту не является привилегией только языка Eiffel, этот метод **можно применять** и в любом другом языке программирования.

Главной идеей проектирования по контракту является понятие утверждения. **Утверждение** (assertion) – это булево высказывание, которое никогда не должно принимать ложное значение и поэтому может быть ложным только в результате ошибки. Обычно утверждение проверяется только во время отладки и не проверяется в режиме выполнения. Действительно при выполнении программы никогда не следует предполагать, что утверждение проверяется.

В методе проектирования по контракту определены утверждения трех типов: предусловия, постусловия и инварианты. Предусловия и постусловия применяются к операциям. **Постусловие** – это высказывание относительно того, как будет выглядеть окружающий мир после выполнения операции. Например, если мы определяем для числа операцию «извлечь квадратный корень», постусловие может принимать форму *input = result * result*, где *result* является выходом, а *input* – исходное значение числа. Постусловие – это хороший способ выразить, что должно быть сделано, не говоря при этом, как это сделать. Другими словами, постусловия позволяют отделить интерфейс от реализации.

Предусловие – это высказывание относительно того, как должен выглядеть окружающий мир до выполнения операции. Для операции «извлечь квадратный корень» можно определить предусловие *input >= 0*. Такое предусловие утверждает, что применение операции «извлечь квадратный корень» для отрицательного числа является ошибочным и последствия такого применения не определены.

На первый взгляд эта идея кажется неудачной, поскольку нам придется выполнить некоторые дополнительные проверки, чтобы убедиться в корректности выполнения операции «извлечь квадратный корень». При этом возникает важный вопрос: на кого ляжет ответственность за выполнение этой проверки.

Предусловие явным образом устанавливает, что за подобную проверку отвечает вызывающий объект. Без такого явного указания обязанностей мы можем получить либо недостаточный уровень проверки (когда каждая из сторон предполагает, что ответственность несет другая сторона), либо чрезмерную проверку (когда она будет выполняться обеими сторонами). Излишняя проверка тоже плоха, поскольку это влечет за собой дублирование кода проверки, что, в свою очередь, может существенно увеличить сложность про-

граммы. Явное определение ответственности помогает снизить сложность кода. Опасность того, что вызывающий объект забудет выполнить проверку, уменьшается тем обстоятельством, что утверждение обычно проверяется во время отладки и тестирования.

Исходя из этих определений предусловия и постусловия, мы можем дать строгое определение термина **исключение**. Исключение возникает, когда предусловие операции выполнено, но операция не может возвратить значение в соответствии с указанным постусловием.

Инвариант представляет собой утверждение относительно класса. Например, класс Account (Счет) может иметь инвариант, который утверждает, что *balance == sum(entries.amount( ))*. Инвариант должен быть «всегда» истинным для всех экземпляров класса. В данном случае «всегда» означает «всякий раз, когда объект доступен для выполнения над ним операции».

По существу это означает, что инвариант дополняет предусловия и постусловия, связанные со всеми открытыми операциями данного класса. Значение инварианта может оказаться ложным во время выполнения некоторого метода, однако оно должно снова стать истинным к моменту взаимодействия с любым другим объектом.

Утверждения могут играть уникальную роль в определении подклассов. Одна из опасностей наследования состоит в том, что операции подкласса можно переопределить так, что они станут не совместимыми с операциями суперкласса. Утверждения уменьшают вероятность этого. Инварианты и постусловия класса должны применяться ко всем подклассам. Подклассы могут усилить эти утверждения, но не могут их ослабить. С другой стороны, предусловия нельзя усилить, но можно ослабить.

На первый взгляд все это кажется излишним, однако имеет весьма важное значение для обеспечения динамического связывания. В соответствии с принципом замещения необходимо всегда иметь возможность обратиться к объекту подкласса так, как если бы он был экземпляром суперкласса. Если подкласс усилил свое предусловие, то операция суперкласса, примененная к подклассу, может завершиться аварийно.

Где найти дополнительную информацию

Все упомянутые мной в главе 1 книги по основам UML рассказывают о диаграммах классов более подробно. Управление зависимостями является критическим элементом больших проектов. Лучшая книга по этой теме – [30].

4

Диаграммы последовательности

Диаграммы взаимодействия (interaction diagrams) описывают взаимодействие групп объектов в различных условиях их поведения. UML определяет диаграммы взаимодействия нескольких типов, из которых наиболее употребительными являются диаграммы последовательности.

Обычно диаграмма последовательности описывает один сценарий. На диаграмме показаны экземпляры объектов и сообщения, которыми обмениваются объекты в рамках одного прецедента (use case).

Для того чтобы начать обсуждение, рассмотрим простой сценарий. Предположим, что у нас есть заказ, и мы собираемся вызвать команду для определения его стоимости. При этом объекту заказа (`Order`) необходимо просмотреть все позиции заказа (`Line Items`) и определить их цены, основанные на правилах построения цены продукции в строке заказа (`Order Line`). Проделав это для всех позиций заказа, объект заказа должен вычислить общую скидку, которая определяется индивидуально для каждого клиента.

На рис. 4.1 приведена диаграмма, представляющая реализацию данного сценария. Диаграммы последовательности показывают взаимодействие, представляя каждого участника вместе с его линией жизни (lifeline), которая идет вертикально вниз и упорядочивает сообщения на странице; сообщения также следует читать сверху вниз.

Одно из преимуществ диаграммы последовательности заключается в том, что мне почти не придется объяснять ее нотацию. Можно видеть, что экземпляр заказа посылает строке заказа сообщения `getQuantity` и `getProduct`. Можно также видеть, как заказ применяет метод к самому себе и как этот метод посылает сообщение `getDiscountInfo` экземпляру клиента.

Однако диаграмма не все показывает так хорошо. Последовательность сообщений `getQuantity`, `getProduct`, `getPricingDetails` и `calculateBasePrice` должна быть реализована для каждой строки заказа, тогда как метод `calculateDiscounts` вызывается лишь однажды. Такое заключение нель-

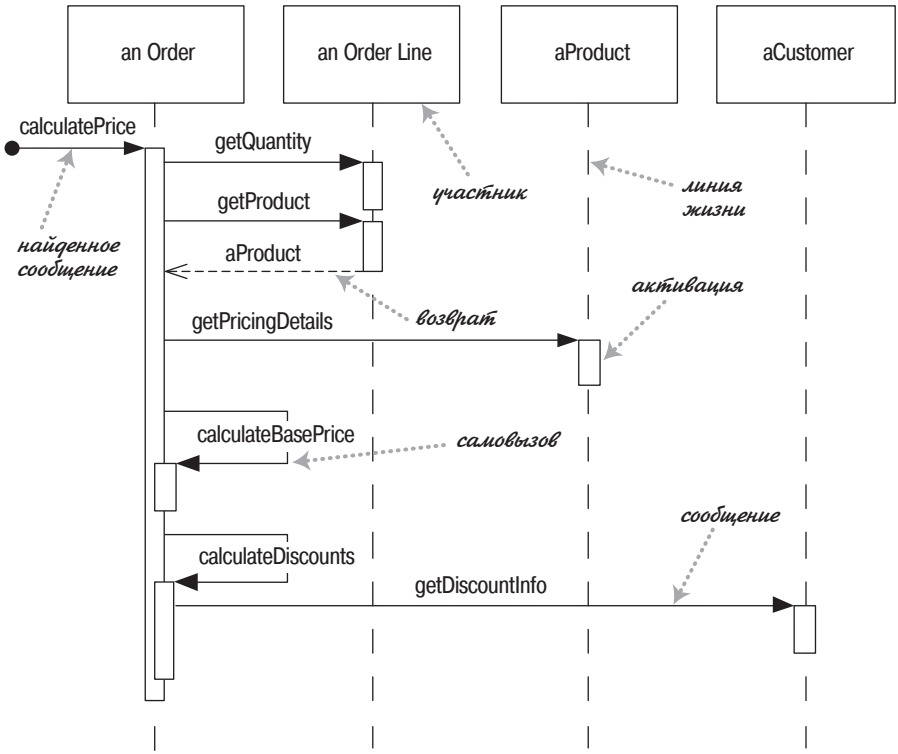

Рис. 4.1. *Диаграмма последовательности централизованного управления*

зя сделать на основе этой диаграммы, но позднее я введу дополнительное обозначение, которое поможет в этом.

В большинстве случаев можно считать участников диаграммы взаимодействия объектами, как это и было в действительности в UML 1. Но в UML 2 их роль значительно сложнее, и полное ее объяснение выходит за рамки этой книги. Поэтому я употребляю термин **участники** (participants), который формально не входит в спецификацию UML. В UML версии 1 участники были объектами, и поэтому их имена подчеркивались, но в UML 2 их надо показывать без подчеркивания, как я и сделал выше.

На приведенной диаграмме я именовал участников, используя стиль `anOrder`. В большинстве случаев это вполне приемлемо. Вот более полный синтаксис: `имя : Класс`, где и имя, и класс не обязательны, но если класс используется, то двоеточие должно присутствовать. (Этот стиль выдержан на рис. 4.4.)

Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника. В языке UML полосы активности не обязательны, но я считаю их исключи-

тельно удобными при пояснении поведения. Единственным исключением является стадия проработки дизайна, поскольку их неудобно рисовать на белых досках.

Именование бывает часто полезным для установления связей между участниками на диаграмме. Как видно на диаграмме, вызов метода `getProduct` возвращает `aProduct`, имеющего то же самое имя и, следовательно, означающего того же самого участника, `aProduct`, которому посылается вызов `getPricingDetails`. Обратите внимание, что обратной стрелкой я обозначил только этот вызов с целью показать соответствие. Многие разработчики используют возвраты для всех вызовов, но я предпочитаю применять их, только когда это дает дополнительную информацию; в противном случае они просто вносят неразбериху. Не исключено, что даже в данном случае можно было опустить возврат, не запутав читателя.

У первого сообщения нет участника, пославшего его, поскольку оно приходит из неизвестного источника. Оно называется **найденным сообщением** (found message).

Другой подход можно увидеть на рис. 4.2. Основная задача остается той же самой, но способ взаимодействия участников для ее решения совершенно другой. Заказ спрашивает каждую строку заказа о его собственной цене (Price). Сама строка заказа передает вычисление дальше – объекту продукта (Product); обратите внимание, как мы показываем передачу параметра. Подобным же образом для вычисления скидки объект заказа вызывает метод для клиента (Customer). Поскольку для выполнения этой задачи клиенту требуется информация от объекта заказа, то он делает повторный вызов в отношении заказа для получения этих данных.

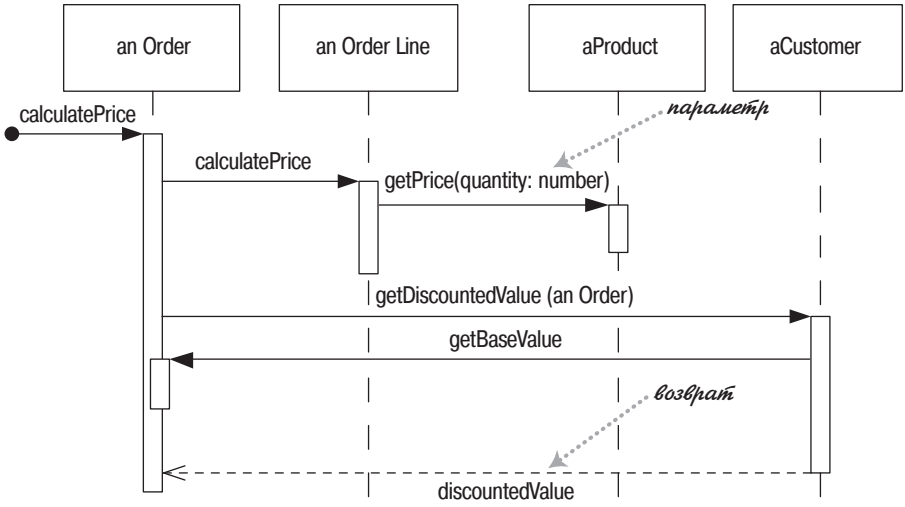

Рис. 4.2. *Диаграмма последовательности для распределенного управления*

Во-первых, на этих двух диаграммах надо обратить внимание на то, насколько ясно диаграмма последовательности показывает различия во взаимодействии участников. В этом проявляется мощь диаграмм взаимодействий. Они не очень хорошо представляют детали алгоритмов, такие как циклы или условное поведение, но делают абсолютно прозрачными вызовы между участниками и дают действительно ясную картину того, какую обработку выполняют конкретные участники.

Во-вторых, посмотрите, как четко видна разница в стиле между двумя взаимодействиями. На рис. 4.1 представлено **централизованное управление** (centralized control), когда один из участников в значительной степени выполняет всю обработку, а другие предоставляют данные. На рис. 4.2 изображено **распределенное управление** (distributed control), при котором обработка распределяется между многими участниками, каждый их которых выполняет небольшую часть алгоритма.

Оба стиля обладают преимуществами и недостатками. Большинство разработчиков, особенно новички в объектно-ориентированном программировании, чаще всего применяют централизованное управление. Во многих случаях это проще, так как вся обработка сосредоточена в одном месте; напротив, в случае распределенного управления при попытке понять программу создается ощущение погони за объектами.

Несмотря на это фанатики объектов, такие как я, предпочитают распределенное управление. Одна из главных задач хорошего проектирования заключается в локализации изменений. Данные и программный код, получающий доступ к этим данным, часто изменяются вместе. Поэтому размещение данных и обращающейся к ним программы в одном месте – первое правило объектно-ориентированного проектирования.

Кроме того, распределенное управление позволяет создать больше возможностей для применения полиморфизма, чем в случае применения условной логики. Если алгоритмы определения цены отличаются для различных типов продуктов, то механизм распределенного управления позволяет нам использовать подклассы класса продукта (Product) для обработки этих вариантов.

Вообще, объектно-ориентированный стиль предназначен для работы с большим количеством небольших объектов, обладающих множеством небольших методов, что дает широкие возможности для переопределения и изменения. Этот стиль сбивает с толку людей, применяющих длинные процедуры; действительно это изменение является сердцем **смены парадигмы** (paradigm shift) при объектной ориентации. Научить этому трудно. Представляется, что единственный способ действительно понять это заключается в использовании распределенного управления при работе в объектно-ориентированном окружении. Многие люди говорят, что они испытали внезапное озарение, когда поняли смысл этого стиля. В этот момент их мозг перестроился, и они начали думать, что децентрализованное управление действительно проще.

Создание и удаление участников

В диаграммах последовательности для создания и удаления участников применяются некоторые дополнительные обозначения (рис. 4.3). В случае создания участника надо нарисовать стрелку сообщения, направленную к прямоугольнику участника. Если применяется конструктор, то имя сообщения не обязательно, но я обычно маркирую его словом «new» в любом случае. Если участник выполняет что-нибудь непосредственно после создания, например команду запроса, то надо начать активацию сразу после прямоугольника участника.

Удаление участника обозначается большим крестом (X). Стрелка сообщения, идущая в X, означает, что один участник явным образом удаляет другого; X в конце линии жизни показывает, что участник удаляет сам себя.

Если в системе работает сборщик мусора, то объекты не удаляются вручную, тем не менее следует при помощи X показать, что объект больше не нужен и готов к удалению. Так следует поступать и в случае операций закрытия, показывая, что объект больше не используется.

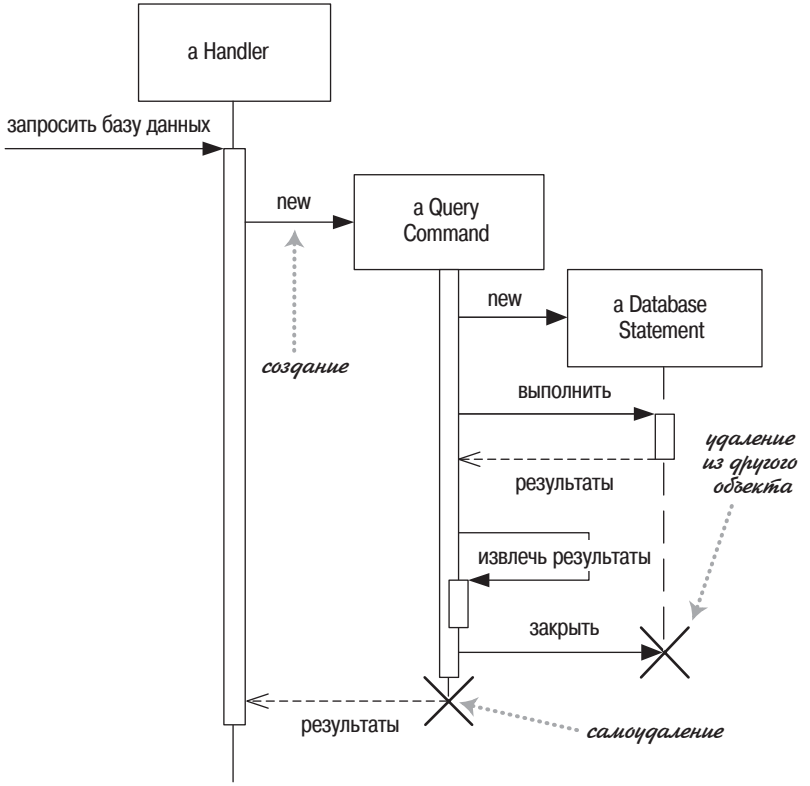

Рис. 4.3. *Создание и удаление участников*

Циклы, условия и тому подобное

Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции. Прежде всего надо усвоить, что диаграммы последовательности для этого не предназначены. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода. Диаграммы последовательности применяются для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления.

Как было сказано, существуют дополнительные обозначения. И для циклов, и для условий используются **фреймы взаимодействий** (interaction frames), представляющие собой средство разметки диаграммы взаимодействия. На рис. 4.4 показан простой алгоритм, основанный на следующем псевдокоде.

```
foreach (lineitem)
  if (product.value > $10K)
    careful.dispatch
  else
    regular.dispatch
  end if
```

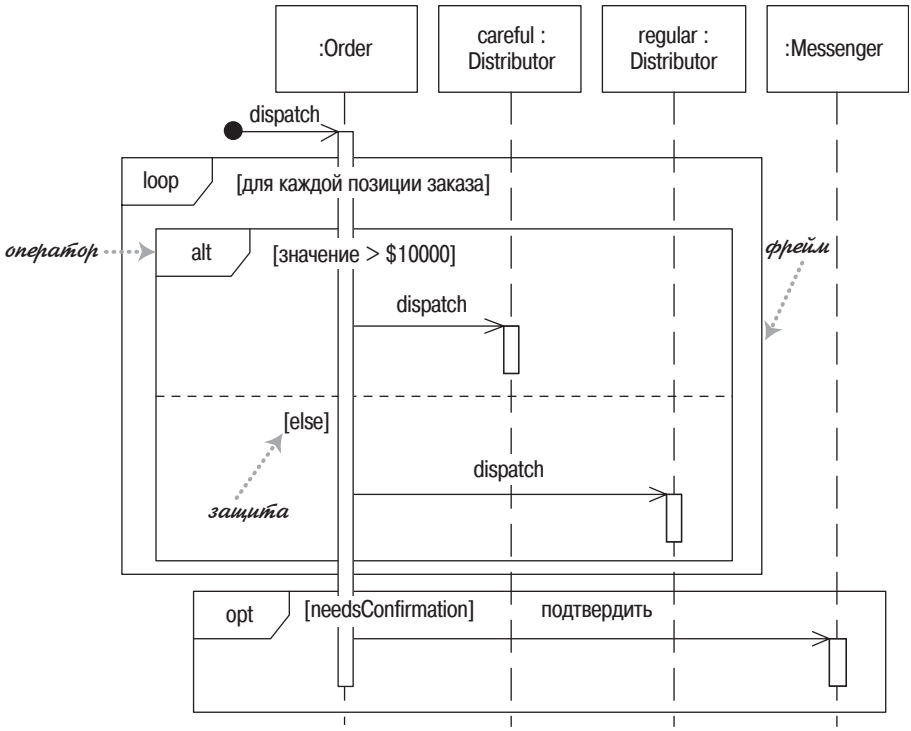

Рис. 4.4. *Фреймы взаимодействия*

```
    end for
    if (needsConfirmation) messenger.confirm
  end procedure
```

В основном фреймы состоят из некоторой области диаграммы последовательности, разделенной на несколько фрагментов. Каждый фрейм имеет оператор, а каждый фрагмент может иметь защиту. (В табл. 4.1 перечислены общепринятые операторы для фреймов взаимодействия.) Для отображения цикла применяется оператор `loop` с единственным фрагментом, а тело итерации помещается в защиту. Для условной логики можно использовать оператор `alt` и помещать условие в каждый фрагмент. Будет выполнен только тот фрагмент, защита которого имеет истинное значение. Для единственной области существует оператор `opt`.

Таблица 4.1. Общепринятые операторы для фреймов взаимодействия

Оператор	Значение
`alt`	Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно (рис. 4.4)
`opt`	Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно `alt` с одной веткой (рис. 4.4)
`par`	Параллельный (parallel); все фрагменты выполняются параллельно
`loop`	Цикл (loop); фрагмент может выполняться несколько раз, а защита обозначает тело итерации (рис. 4.4)
`region`	Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием
`neg`	Отрицательный (negative) фрагмент; обозначает неверное взаимодействие
`ref`	Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение
`sd`	Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо

Фреймы взаимодействия – новинка UML 2. В диаграммах, разработанных до создания UML 2, применяется другой подход; кроме того, некоторые разработчики не любят фреймы и предпочитают прежние соглашения. На рис. 4.5 показаны некоторые из этих неофициальных приемов.

В UML 1 использовались маркеры итераций и защиты. В качестве **маркера итерации** (iteration marker) выступал символ *, добавленный к имени сообщения. Для обозначения тела итерации можно добавить текст в квадратных скобках. **Защита** (guard) – это условное выраже-

ние, размещенное в квадратных скобках и означающее, что сообщение посылается, только когда защита принимает истинное значение. Эти обозначения исключены из UML 2, но они все еще встречаются в диаграммах взаимодействия.

Несмотря на то что маркеры итерации и защиты могут оказаться полезными, они имеют один недостаток. С помощью защиты нельзя показать, что несколько защит взаимно исключают друг друга, например две защиты, представленные на рис. 4.5. Оба обозначения работают только в случае отправки одного сообщения и не работают, когда при одной активации посылается несколько сообщений в рамках того же самого цикла или условного блока.

Решением последней проблемы может служить ставшее популярным неофициальное соглашение, заключающееся в применении **псевдосообщения** (pseudomessage) в виде условия цикла или защиты на одном из вариантов обозначения самовызова. На рис. 4.5 я показал это без стрелки сообщения; некоторые разработчики включают стрелку сообщения, но ее отсутствие помогает подчеркнуть, что это ненастоящий вызов. Некоторые разработчики любят оттенять прямоугольник активации псевдосообщения серым цветом. Вариативное поведение можно показать, поставив маркер альтернативы между активациями.

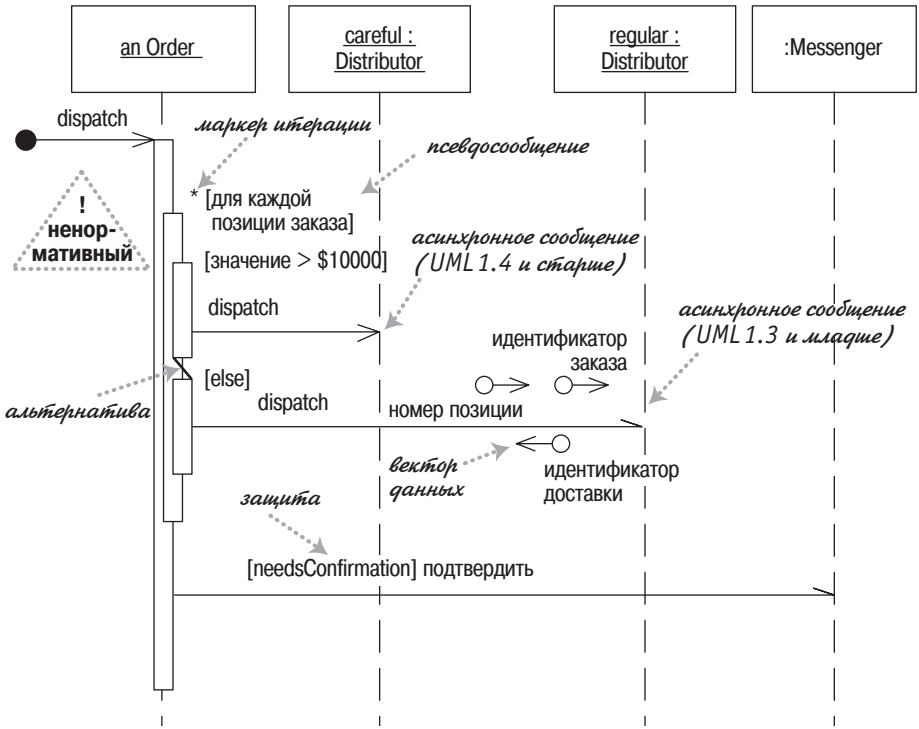

Рис. 4.5. *Старые соглашения для условной логики*

Хотя я считаю активации очень полезными, они не слишком много дают в случае метода dispatch (перенаправить), с помощью которого отправляются сообщения, при этом в рамках активации приемника больше ничего не происходит. По общепринятому соглашению, которого я придерживался в диаграмме на рис. 4.5, для таких простых вызовов активация опускается.

Стандарт UML не предоставляет графических средств для обозначения передаваемых данных; вместо этого они показываются с помощью параметров в имени сообщения и на стрелках возврата. **Векторы данных** повсеместно использовались во многих областях для обозначения перемещения данных, и многие разработчики все еще с удовольствием применяют их в UML.

В конечном счете, хотя различные системы могут включать в диаграммы последовательности обозначения для условной логики, я не думаю, что они работают сколько-нибудь лучше программного кода или, по крайней мере, псевдокода. В частности, я считаю фреймы взаимодействия очень тяжеловесными, скрывающими основной смысл диаграммы, поэтому я предпочитаю псевдосообщения.

Синхронные и асинхронные вызовы

Если вы были очень внимательны, то заметили, что стрелки в последних двух диаграммах отличаются от предыдущих. Это небольшое отличие достаточно важно в UML версии 2. Здесь закрашенные стрелки показывают синхронное сообщение, а простые стрелки обозначают асинхронное сообщение.

Если вызывающий объект посылает **синхронное сообщение** (synchronous message), то он должен ждать, пока обработка сообщения не будет закончена, например при вызове подпрограммы. Если вызывающий объект посылает **асинхронное сообщение** (asynchronous message), то он может продолжать работу и не должен ждать ответа. Асинхронные вызовы можно встретить в многопоточных приложениях и в промежуточном программном обеспечении, ориентированном на сообщения. Асинхронность улучшает способность к реагированию и уменьшает количество временных соединений, но сложнее в отладке.

Разница в изображении стрелок едва уловима; действительно их довольно трудно отличить. Кроме того, это изменение, введенное в UML 1.4, не обладает обратной совместимостью, поскольку до этого асинхронные сообщения обозначались половинными стрелками, как показано на рис. 4.5.

На мой взгляд, такое различие слишком незаметно. Я бы советовал выделять асинхронные сообщения при помощи старых половинных стрелок, которые больше привлекают взгляд. Читая диаграмму последовательности, не спешите делать предположения о синхронности по

виду стрелок до тех пор, пока не убедитесь, что автор умышленно нарисовал их разными.

Когда применяются диаграммы последовательности

Диаграммы последовательности следует применять тогда, когда требуется посмотреть на поведение нескольких объектов в рамках одного прецедента. Диаграммы последовательности хороши для представления взаимодействия объектов, но не очень подходят для точного определения поведения.

Если вы хотите посмотреть на поведение одного объекта в нескольких прецедентах, то примените диаграмму состояния (глава 10). Если же надо изучить поведение нескольких объектов в нескольких прецедентах или потоках, не забудьте о диаграмме деятельности (глава 11).

Если требуется быстро исследовать несколько вариантов взаимодействия, лучше использовать CRC-карточки, поскольку это позволяет избежать непрерывного рисования и стирания. Часто бывает удобно поработать с CRC-карточками для просмотра вариантов взаимодействия, а затем с помощью диаграмм взаимодействий фиксировать те взаимодействия, которые будут применяться позже.

Другим полезным видом диаграмм взаимодействий являются коммуникационные диаграммы, которые показывают соединения, и временные диаграммы, показывающие временные интервалы.

CRC–карточки

Одним из наиболее полезных приемов, соответствующих хорошему стилю ООП, является исследование взаимодействия объектов, поскольку его цель состоит в том, чтобы исследовать работу программы, а не данные. CRC-диаграммы (Class-Responsibility-Collaboration, класс-обязанность-кооперация), придуманные Уордом Каннингемом (Ward Cunningham) в конце 80-х годов, выдержали проверку временем и стали высокоэффективным инструментом решения этой задачи (рис. 4.6). И хотя они не входят в состав UML, все же являются очень популярными среди высококвалифицированных разработчиков в области объектных технологий.

Для использования CRC-карточек вы и ваши коллеги должны собраться за столом. Возьмите различные сценарии и проиграйте их с помощью карточек, поднимая их над столом, в то время когда они активны, и передавая их по кругу в предположении, что они посылают сообщение. Эту технологию почти невозможно описать в книге, но легко продемонстрировать; лучший способ научиться этому состоит в том, чтобы попросить кого-нибудь, кто имеет такой опыт, показать вам это.

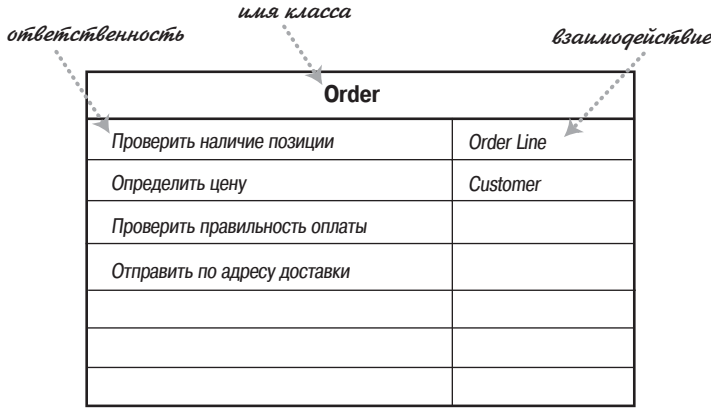

Рис. 4.6. *Пример CRC-карточки*

Важным моментом в CRC-методике является определение ответственностей. **Ответственность** (responsibility) – это краткое описание того, что объект должен делать: операция, которую выполняет объект, некоторый объем знаний, который объект поддерживает, или какие-либо важные решения, принимаемые объектом. Идея состоит в том, чтобы вы могли взять любой класс и сформулировать его разумно ограниченные обязанности. Такой образ действия поможет вам яснее представить себе архитектуру классов.

Вторая буква «С» (в CRC) означает **взаимодействие** (collaboration): другие классы, с которыми должен работать рассматриваемый класс. Это дает вам некоторое представление о связях между классами, но все еще на высоком уровне.

Одно из главных преимуществ CRC-карточек состоит в том, что они способствуют живому обсуждению проектов в среде разработчиков. Если в процессе работы над шаблоном поведения вы хотите посмотреть, как классы его реализуют, то вычерчивание диаграмм взаимодействия, описанных в этой главе, может занять слишком много времени. Обычно требуется просмотреть варианты, а рисование и стирание вариантов на диаграммах может быть очень утомительным. С помощью CRC-карточек разработчики моделируют взаимодействие, поднимая карточки и передавая их по кругу. Это позволяет быстро просчитать варианты.

В процессе работы вы создаете представление об ответственностях и записываете их на карточки. Размышления об ответственностях важны, поскольку рассеивают представление о классах как о бессловесных хранителях данных и помогают членам команды лучше понять поведение каждого класса на высшем уровне. Ответственность может соответствовать операции, атрибуту или, что более точно, неограниченной группе атрибутов и операций.

Мой опыт показывает, что распространенной ошибкой разработчиков является создание ими длинных списков низкоуровневых ответственностей. Это приводит к непониманию сути. Ответственности должны свободно помещаться на одной карточке. Спросите себя, следует ли разделить класс или лучше отрегулировать ответственности, переместив их на более высокий уровень операторов.

Многие разработчики подчеркивают важность ролевой игры, когда каждый член команды играет роль одного или нескольких классов. Я никогда не видел, чтобы Уорд Каннингем так поступал, и думаю, что ролевая игра находится в начале своего пути.

CRC были посвящены целые книги, но не думаю, что они действительно стали сердцем методологии. Первой работой по CRC была статья, написанная Кентом Беком (Kent Beck) [4]. Дополнительную информацию по CRC-карточкам можно найти в [44].