Глава 1. Основные сведения о языке UML

Самое лучшее средство – это большая диаграмма, приколотая к стене.
Даг Скотт

1.1. Цели и история создания языка UML

Унифицированный язык моделирования UML (Unified Modeling Language) – это преемник того поколения методов объектно-ориентированного анализа и проектирования, которые появились в конце 80-х и начале 90-х годов. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо начали работу по объединению их методов Booch [Буч-1999] и OMT (Object Modeling Technique) под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же в 1995 г. к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Буча, Рамбо и Якобсона, однако дополняет их новыми возможностями.

UML находится в процессе стандартизации, проводимом консорциумом OMG (Object Management Group), в настоящее время он принят в качестве стандартного языка моделирования и получил широкую поддержку. UML принят на вооружение практически всеми крупнейшими компаниями – производителями программного обеспечения (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, практически все мировые производители CASE-средств, помимо Rational Software (Rational Rose), поддерживают UML в своих продуктах (Paradigm Plus (CA), System Architect (Popkin Software), Microsoft Visual Modeler и др.). Полное описание UML можно найти на сайтах http://www.omg.org и http://www.rational.com. Первое описание UML на русском языке содержится в книге [Фаулер-1999], в дальнейшем изложении терминология языка соответствует данному переводу. Кроме него, имеются также переводы [Боггс-2000], [Буч-2000] и [Ларман-2001].

1.2. Средства UML

Создатели UML представляют его как язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов. Стандарт UML версии 1.1, принятый OMG в 1997 г., предлагает следующий набор диаграмм для моделирования:

- **диаграммы вариантов использования (use case diagrams)** – для моделирования бизнес-процессов организации и требований к создаваемой системе);
- **диаграммы классов (class diagrams)** – для моделирования статической структуры классов системы и связей между ними;
- **диаграммы поведения системы (behavior diagrams):**
  - **диаграммы взаимодействия (interaction diagrams):**
    - **диаграммы последовательности (sequence diagrams)** и
    - **кооперативные диаграммы (collaboration diagrams)** – для моделирования процесса обмена сообщениями между объектами;
  - **диаграммы состояний (statechart diagrams)** – для моделирования поведения объектов системы при переходе из одного состояния в другое;
  - **диаграммы деятельностей (activity diagrams)** – для моделирования поведения системы в рамках различных вариантов использования, или моделирования деятельностей;
- **диаграммы реализации (implementation diagrams):**
  - **диаграммы компонентов (component diagrams)** – для моделирования иерархии компонентов (подсистем) системы;
  - **диаграммы размещения (deployment diagrams)** – для моделирования физической архитектуры системы.

1.3. Диаграммы вариантов использования

Понятие варианта использования (use case) впервые ввел

Ивар Якобсон и придал ему такую значимость, что в настоящее время вариант использования превратился в основной элемент разработки и планирования проекта.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать.

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования.

Действующие лица делятся на три основных типа – пользователи системы, другие системы, взаимодействующие с данной, и время. Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Для наглядного представления вариантов использования в качестве основных элементов процесса разработки программного обеспечения (ПО) применяются диаграммы вариантов использования. На рис. 1.1 показан пример такой диаграммы для банкомата (Automated Teller Machine, ATM).

На данной диаграмме человеческие фигурки обозначают действующих лиц, овалы – варианты использования, а линии и стрелки – различные связи между действующими лицами и вариантами использования.

На этой диаграмме показаны два действующих лица: клиент и кредитная система. Существует также шесть основных действий, выполняемых моделируемой системой: перевести деньги, сделать вклад,

снять деньги со счета, показать баланс, изменить идентификационный код и осуществить оплату.

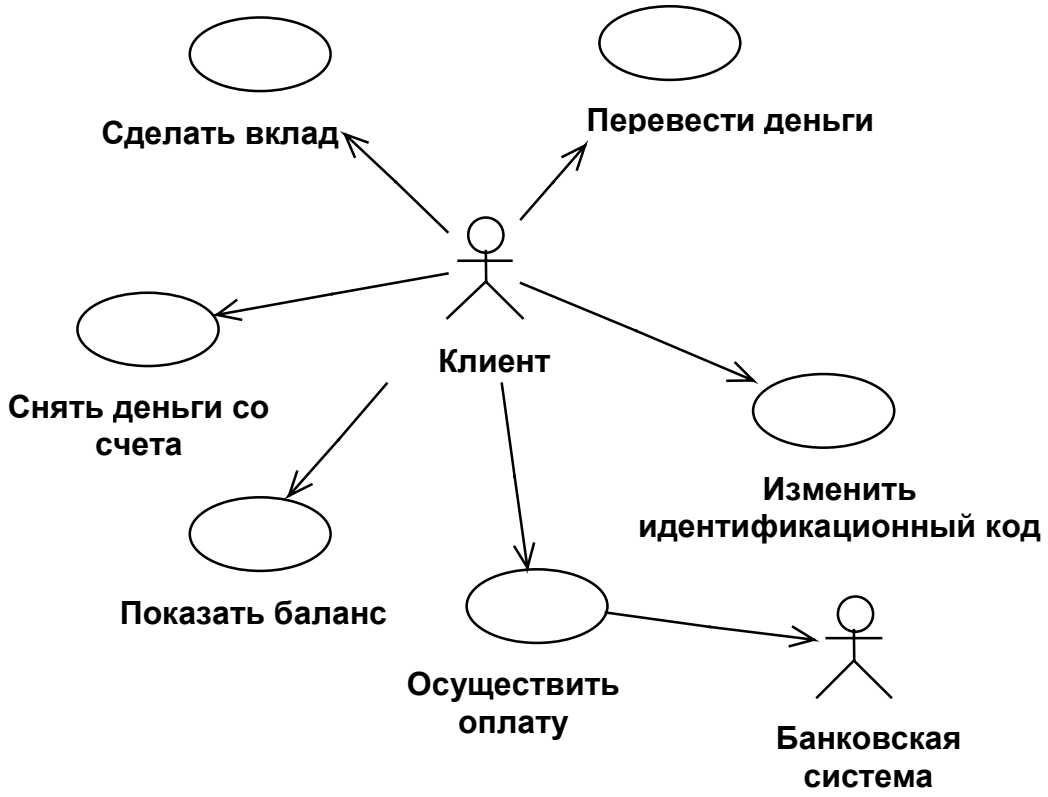

Рис. 1.1. Пример диаграммы вариантов использования

На диаграмме вариантов использования показано взаимодействие между вариантами использования и действующими лицами. Она отражает требования к системе с точки зрения пользователя. Таким образом, варианты использования – это функции, выполняемые системой, а действующие лица – это заинтересованные лица (stakeholders) по отношению к создаваемой системе. Такие диаграммы показывают, какие действующие лица инициируют варианты использования. Из них также видно, когда действующее лицо получает информацию от варианта использования. Данная диаграмма, например, отражает взаимодействие между вариантами использования и действующими лицами системы АТМ. В сущности, диаграмма вариантов использования иллюстрирует требования к системе. В нашем примере, клиент банка инициирует большое количество различных вариантов использования: «Снять деньги

со счета», «Перевести деньги», «Сделать вклад», «Показать баланс» и «Изменить идентификационный код». От варианта использования «Осуществить оплату» стрелка указывает на Банковскую систему. Действующими лицами могут быть внешние системы, и потому в данном случае Банковская система показана именно как действующее лицо – она внешняя для системы АТМ. Направленная от варианта использования к действующему лицу стрелка показывает, что вариант использования предоставляет некоторую информацию, используемую действующим лицом. В данном случае вариант использования «Осуществить оплату» предоставляет Банковской системе информацию об оплате по кредитной карточке.

Все варианты использования, так или иначе, связаны с внешними требованиями к функциональности системы. Варианты использования всегда следует анализировать вместе с действующими лицами системы, определяя при этом реальные задачи пользователей и рассматривая альтернативные способы решения этих задач.

Действующие лица могут играть различные роли по отношению к варианту использования. Они могут пользоваться его результатами или могут сами непосредственно в нем участвовать. Значимость различных ролей действующего лица зависит от того, каким образом используются его связи.

Конкретная цель диаграмм вариантов использования – это документирование вариантов использования (всё, входящее в сферу применения системы), действующих лиц (всё вне этой сферы) и связей между ними. Разрабатывая диаграммы вариантов использования, старайтесь придерживаться следующих правил:

- Не моделируйте связи между действующими лицами. По определению действующие лица находятся вне сферы действия системы. Это означает, что связи между ними также не относятся к её компетенции.
- Не соединяйте сплошной стрелкой (коммуникационной связью) два варианта использования непосредственно. Диаграммы данного типа описывают только, какие варианты использования доступны системе, а не порядок их выполнения. Для отображения порядка

выполнения вариантов использования применяют диаграммы деятельности.

– Вариант использования должен быть инициирован действующим лицом. Это означает, что должна быть сплошная стрелка, начинающаяся на действующем лице и заканчивающаяся на варианте использования.

Хорошим источником для идентификации вариантов использования служат внешние события. Следует начать с перечисления всех событий, происходящих во внешнем мире, на которые система должна каким-то образом реагировать. Какое-либо конкретное событие может повлечь за собой реакцию системы, не требующую вмешательства пользователей, или, наоборот, вызвать пользовательскую реакцию. Идентификация событий, на которые необходимо реагировать, помогает идентифицировать варианты использования.

Варианты использования начинают описывать, что должна будет делать система. Чтобы фактически разработать систему, однако, потребуются более конкретные детали. Эти детали описываются в документе, называемом «поток событий» (flow of events). Целью потока событий является документирование процесса обработки данных, реализуемого в рамках варианта использования. Этот документ подробно описывает, что будут делать пользователи системы, и что – сама система.

Хотя поток событий и описывается подробно, он также не должен зависеть от реализации. Цель – описать, что будет делать система, а не как она будет делать это. Обычно поток событий включает:

– краткое описание;
– предусловия (pre-conditions);
– основной поток событий;
– альтернативный поток событий (или несколько альтернативных потоков);
– постусловия (post-conditions).

Последовательно рассмотрим эти составные части.

Описание

Каждый вариант использования должен иметь связанное с ним короткое описание того, что он будет делать. Например, вариант

использования «Перевести деньги» системы АТМ может содержать следующее описание:

Вариант Использования «Перевести деньги» позволяет клиенту или служащему банка переводить деньги с одного счета до востребования или сберегательного счета на другой.

Предусловия

Предусловия варианта использования – это такие условия, которые должны быть выполнены, прежде чем вариант использования начнет выполняться сам. Например, таким условием может быть выполнение другого варианта использования или наличие у пользователя прав доступа, требуемых для запуска этого. Не у всех вариантов использования бывают предварительные условия.

Ранее упоминалось, что диаграммы вариантов использования не должны отражать порядок их выполнения. С помощью предусловий, однако, можно документировать и такую информацию. Например, предусловием одного варианта использования может быть то, что в это время должен выполняться другой.

Основной и альтернативный потоки событий

Конкретные детали вариантов использования описываются в основном и альтернативных потоках событий. Поток событий поэтапно описывает, что должно происходить во время выполнения заложенной в варианты использования функциональности. Поток событий уделяет внимание тому, что будет делать система, а не как она будет делать это, причем описывает все это с точки зрения пользователя. Основной и альтернативный потоки событий включают следующее описание:

- способ запуска варианта использования;
- различные пути выполнения варианта использования;
- нормальный, или основной, поток событий варианта использования;
- отклонения от основного потока событий (так называемые альтернативные потоки);
- потоки ошибок;
- способ завершения варианта использования.

Например, поток событий варианта использования «Снять деньги» может выглядеть следующим образом:

Основной поток

1. Вариант использования начинается, когда клиент вставляет свою карточку в АТМ.

2. АТМ выводит приветствие и предлагает клиенту ввести свой персональный идентификационный номер.

3. Клиент вводит номер.

4. АТМ подтверждает введённый номер. Если номер не подтвержден, выполняется альтернативный поток событий А1.

5. АТМ выводит список доступных действий:

- положить деньги на счет;
- снять деньги со счета;
- перевести деньги.

6. Клиент выбирает пункт «Снять деньги».

7. АТМ запрашивает, сколько денег надо снять.

8. Клиент вводит требуемую сумму.

9. АТМ определяет, имеется ли на счету достаточно денег. Если денег недостаточно, выполняется альтернативный поток А2. Если во время подтверждения суммы возникают ошибки, выполняется поток ошибок Е1.

10. АТМ вычитает требуемую сумму из счета клиента.

11. АТМ выдает клиенту требуемую сумму наличными.

12. АТМ возвращает клиенту его карточку.

13. АТМ печатает чек для клиента.

14. Вариант использования завершается.

Альтернативный поток А1. Ввод неправильного идентификационного номера.

1. АТМ информирует клиента, что идентификационный номер введён неправильно.

2. АТМ возвращает клиенту его карточку.

3. Вариант использования завершается.

Альтернативный вариант использования А2. Недостаточно денег на счету.

1. АТМ информирует клиента, что денег на его счету недостаточно.

2. АТМ возвращает клиенту его карточку.

3. Вариант использования завершается.

Поток ошибок Е1. Ошибка в подтверждении запрашиваемой суммы.

1. АТМ сообщает пользователю, что при подтверждении запрашиваемой суммы произошла ошибка и дает ему номер телефона службы поддержки клиентов банка.

2. АТМ заносит сведения об ошибке в журнал ошибок. Каждая запись содержит дату и время ошибки, имя клиента, номер его счета и код ошибки.

3. АТМ возвращает клиенту его карточку.

4. Вариант использования завершается.

Постусловия

Постусловиями называются такие условия, которые всегда должны быть выполнены после завершения варианта использования. Например, в конце варианта использования можно пометить флажком какой-нибудь переключатель. Информация такого типа входит в состав постусловий. Как и для предусловий, с помощью постусловий можно вводить информацию о порядке выполнения вариантов использования системы. Если, например, после одного из вариантов использования должен всегда выполняться другой, это можно описать как постусловие. Такие условия имеются не у каждого варианта использования.

Связи между вариантами использования и действующими лицами

В языке UML на диаграммах вариантов использования поддерживается несколько типов связей между элементами диаграммы. Это связи коммуникации (communication), включения (include), расширения (extend) и обобщения (generalization).

Связь коммуникации – это связь между вариантом использования и действующим лицом. На языке UML связи коммуникации показывают с помощью однонаправленной ассоциации (сплошной линии со стрелкой). Направление стрелки позволяет понять, кто инициирует коммуникацию.

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность. В примере АТМ варианты использования «Снять деньги» и «Положить деньги на счет» должны опознать (аутентифицировать) клиента и его идентификационный номер перед тем, как допустить осуществление самой транзакции. Вместо того чтобы подробно описывать процесс аутентификации для каждого из них, можно поместить эту функциональность в свой собственный вариант использования под названием «Аутентифицировать клиента».

Связь расширения применяется при описании изменений в нормальном поведении системы. Она позволяет варианту использования только при необходимости использовать функциональные возможности другого.

На языке UML связи включения и расширения показывают в виде зависимостей с соответствующими стереотипами, как показано на рис. 1.2.

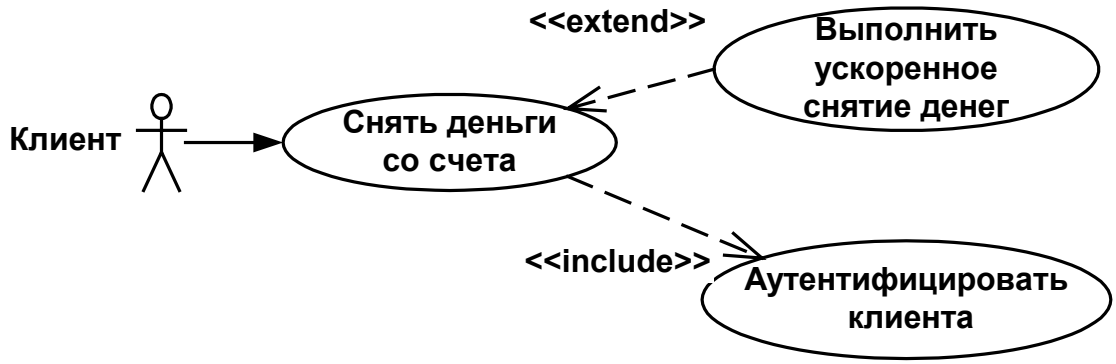

Рис. 1.2. Связи использования и расширения

С помощью связи обобщения показывают, что у нескольких действующих лиц имеются общие черты. Например, клиенты могут быть двух типов: корпоративные и индивидуальные. Эту связь можно моделировать с помощью нотации, показанной на рис. 1.3.

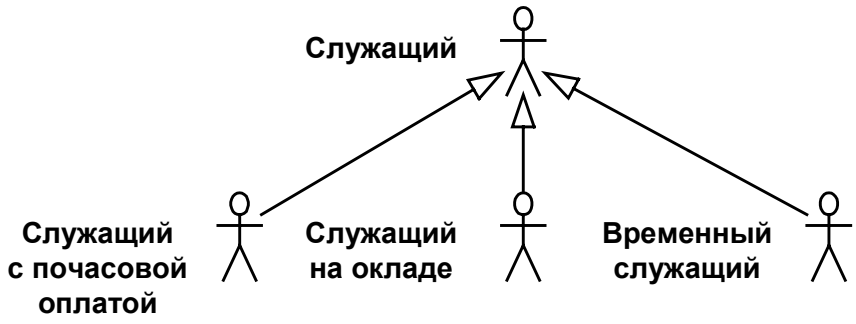

Рис. 1.3. Обобщение действующего лица

Нет необходимости всегда создавать связи этого типа. В общем случае, они нужны, если поведение действующего лица одного типа отличается от поведения другого постольку, поскольку это затрагивает систему. Если оба подтипа используют одни и те же варианты использования, показывать обобщение действующего лица не требуется.

Варианты использования являются необходимым средством на стадии формирования требований к ПО. Каждый вариант использования – это потенциальное требование к системе, и пока оно не выявлено, невозможно запланировать его реализацию.

1.4. Диаграммы взаимодействия

Диаграммы взаимодействия (interaction diagrams) описывают поведение взаимодействующих групп объектов.

Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного варианта использования. На такой диаграмме отображается ряд объектов и те сообщения, которыми они обмениваются между собой.

Сообщение (message) – это средство, с помощью которого объект-отправитель запрашивает у объекта получателя выполнение одной из его операций.

Информационное (informative) сообщение – это сообщение, снабжающее объект-получатель некоторой информацией для обновления его состояния.

Сообщение-запрос (interrogative) – это сообщение, запрашивающее выдачу некоторой информации об объекте-получателе.

Императивное (imperative) сообщение – это сообщение, запрашивающее у объекта-получателя выполнение некоторых действий.

Существует два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams).

1.4.1. Диаграммы последовательности

Диаграммы последовательности отражают поток событий, происходящих в рамках варианта использования. Например, вариант использования «Снять деньги» предусматривает несколько возможных последовательностей, такие как снятие денег, попытка снять деньги, не имея их достаточного количества на счету, попытка снять деньги по неправильному идентификационному номеру и некоторые другие. Нормальный сценарий снятия денег со счета (при отсутствии таких проблем, как неправильный идентификационный номер или недостаток денег на счете) показан на рис. 1.4.

Эта диаграмма последовательности показывает поток событий в рамках варианта использования «Снять деньги». Все действующие лица показаны в верхней части диаграммы; в приведенном выше примере изображено действующее лицо Клиент. Объекты, требуемые системе для выполнения варианта использования «Снять деньги», также представлены в верхней части диаграммы. Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций.

На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия.

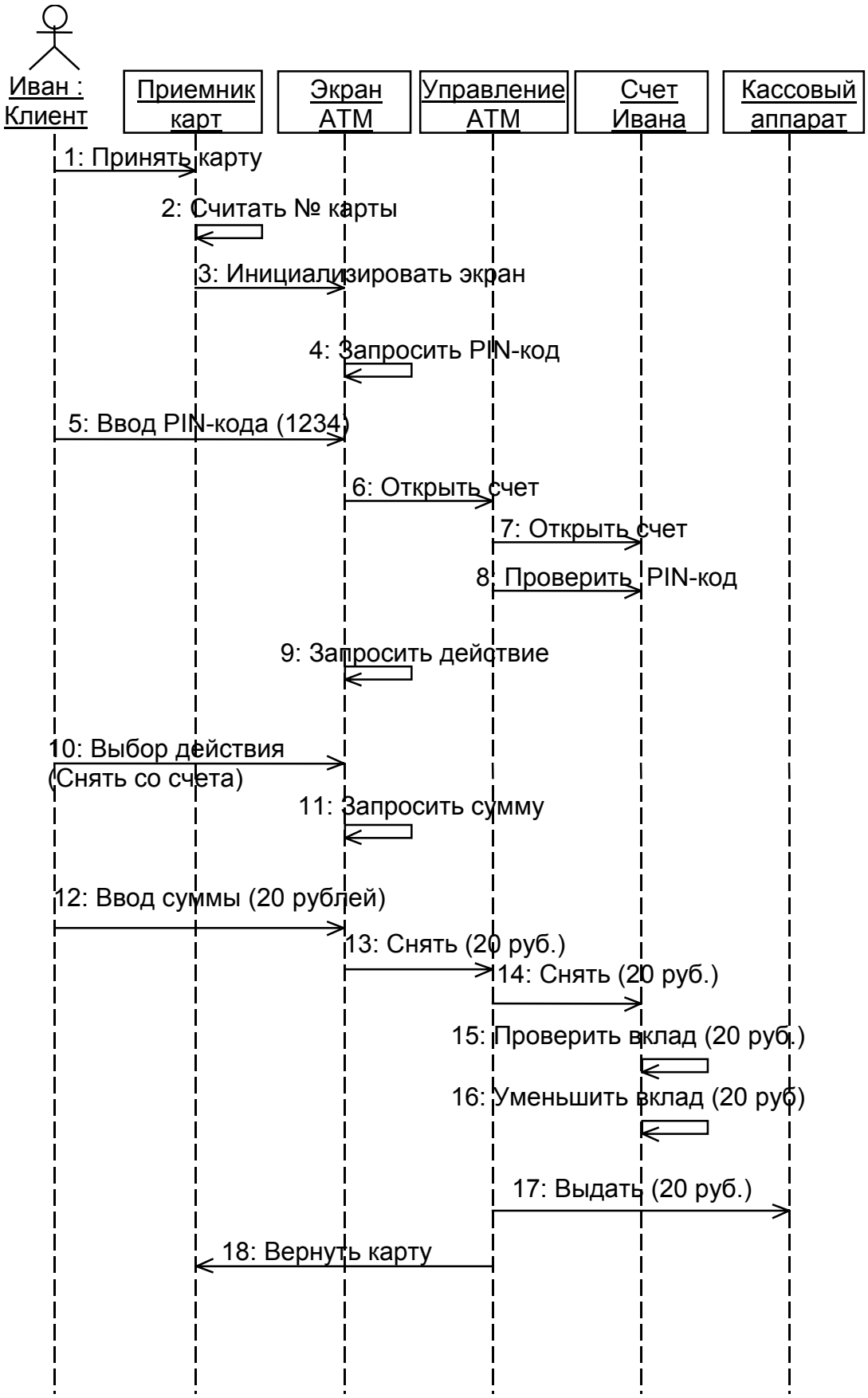

Рис. 1.4. Диаграмма последовательности для снятия клиентом денег со счета

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое сообщение помечается как минимум именем сообщения; при желании можно добавить также аргументы и некоторую управляющую информацию, и, кроме того, можно показать само-делегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни.

Хороший способ первоначального обнаружения некоторых объектов – это изучение имен существительных в потоке событий. Можно также прочитать документы, описывающие конкретный сценарий. Под сценарием понимается конкретный экземпляр потока событий. Поток событий для варианта использования «Снять деньги» говорит о человеке, снимающем некоторую сумму денег со счета с помощью АТМ.

Не все объекты появляются в потоке событий. Там, например, может не быть форм для заполнения, но их необходимо показать на диаграмме, чтобы позволить действующему лицу ввести новую информацию в систему или просмотреть её. В потоке событий, скорее всего, также не будет и управляющих объектов (control objects). Эти объекты управляют последовательностью потока в варианте использования.

1.4.2. Кооперативные диаграммы

Вторым видом диаграммы взаимодействия является кооперативная диаграмма.

Подобно диаграммам последовательности, кооперативные диаграммы (collaborations) отображают поток событий через конкретный сценарий варианта использования. Диаграммы последовательности упорядочены по времени, а кооперативные диаграммы больше внимания заостряют на связях между объектами. На рис. 1.5 приведена кооперативная диаграмма, описывающая, как клиент снимает деньги со счета.

Как видно из рисунка, здесь представлена вся та информация, которая была и на диаграмме последовательности, но кооперативная диаграмма по-другому описывает поток событий. Из нее легче понять связи между объектами, однако, труднее уяснить последовательность событий.

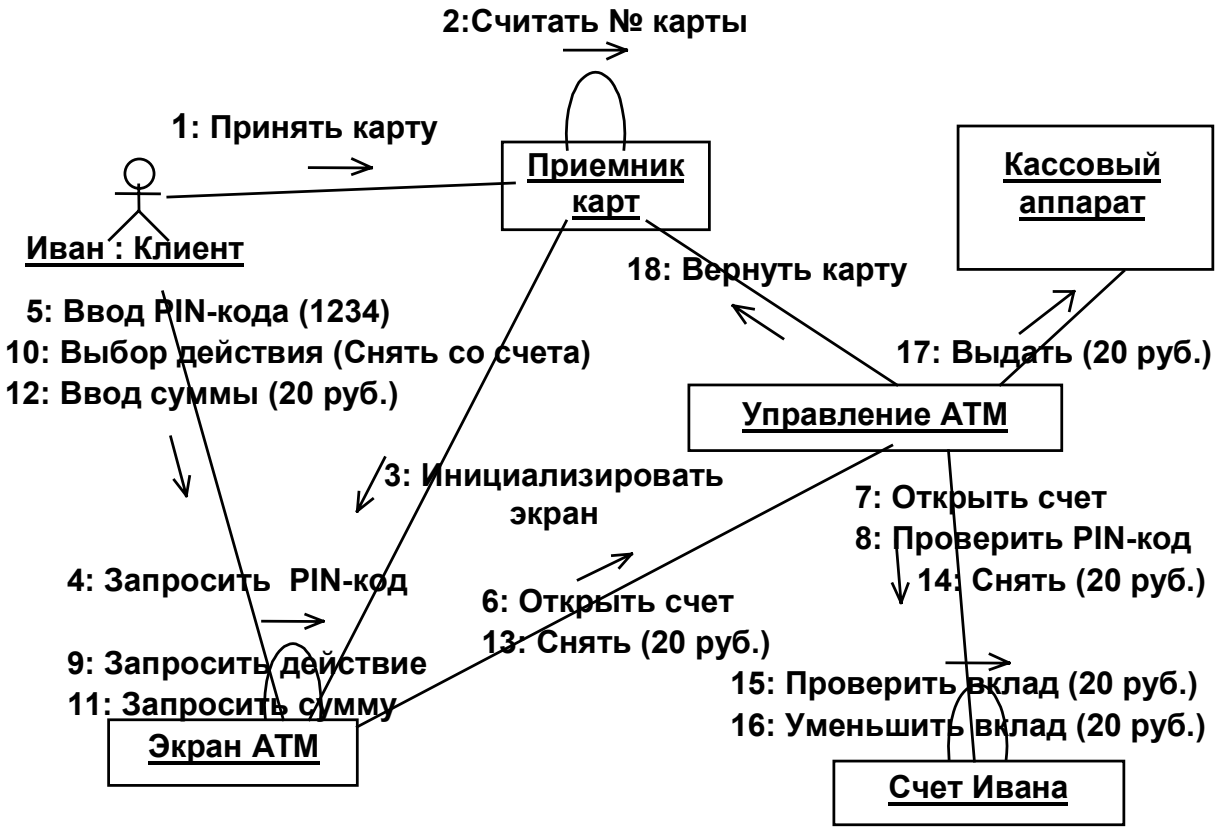

Рис. 1.5. Кооперативная диаграмма, описывающая процесс снятия клиентом денег со своего счета

По этой причине часто для какого-либо сценария создают диаграммы обоих типов. Хотя они служат одной и той же цели и содержат одну и ту же информацию, но представляют ее с различных точек зрения.

На кооперативной диаграмме так же, как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временна́я последовательность, однако, указывается путем нумерации сообщений.

1.5. Диаграммы классов

1.5.1. Общие сведения

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами.

Диаграмма классов для варианта использования «Снять деньги» показана на рис. 1.6.

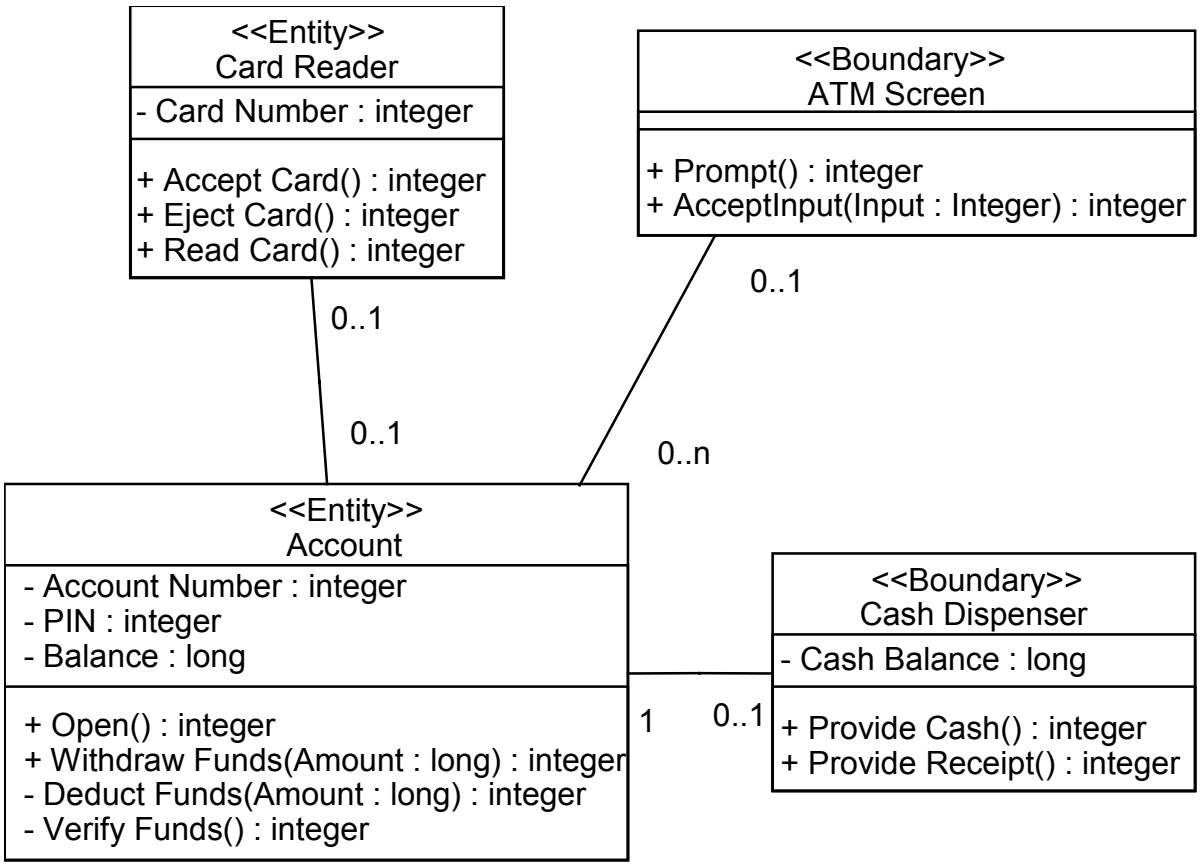

Рис. 1.6. Диаграмма классов для варианта использования «Снять деньги»

На этой диаграмме классов показаны связи между классами, реализующими вариант использования «Снять деньги». В этом процессе задействованы четыре класса: Card Reader[1] (устройство для чтения карточек), Account (счет), ATM Screen (экран ATM) и Cash Dispenser (кассовый аппарат). Каждый класс на диаграмме выглядит в виде прямоугольника, разделенного на три части. В первой содержится имя класса, во второй – его атрибуты. В последней части содержатся операции класса, отражающие его поведение (действия, выполняемые классом).

[1] На диаграммах классов и всех последующих диаграммах используются английские имена, так как только такие имена поддерживаются в языках программирования. Использование русских имен объектов, операций, атрибутов и т. д. сопряжено с большими трудностями, так как CASE-средства их не поддерживают должным образом.

Связывающие классы линии отражают взаимодействие между классами. Так, класс Account связан с классом ATM Screen (экран ATM), потому что они непосредственно сообщаются и взаимодействуют друг с другом. Класс Card Reader (устройство для чтения карточек) не связан с классом Cash Dispenser (кассовый аппарат), поскольку они не сообщаются друг с другом непосредственно.

1.5.2 Стереотипы классов

Стереотипы – это механизм, позволяющий разделять классы на категории. В языке UML определены три основных стереотипа классов: Boundary (граница), Entity (сущность) и Control (управление).

Граничные классы

Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами.

Чтобы найти граничные классы, надо исследовать диаграммы вариантов использования. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.

Классы-сущности

Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Обычно для каждого класса-сущности создают таблицу в базе данных.

Управляющие классы

Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования. Управляющий класс отвечает за координацию, но сам не несет в себе никакой функциональности, так как остальные классы не посылают ему большого количества сообщений. Вместо этого он сам посылает множество сообщений. Управляющий класс

просто делегирует ответственность другим классам, по этой причине его часто называют классом-менеджером.

В системе могут быть и другие управляющие классы, общие для нескольких вариантов использования. Например, может быть класс SecurityManager (менеджер безопасности), отвечающий за контроль событий, связанных с безопасностью. Класс TransactionManager (менеджер транзакций) занимается координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с другими элементами функционирования системы, такими как разделение ресурсов, распределенная обработка данных или обработка ошибок.

Помимо упомянутых выше стереотипов можно создавать и свои собственные.

1.5.3. Механизм пакетов

Пакеты применяют, чтобы сгруппировать классы, обладающие некоторой общностью. Существует несколько наиболее распространенных подходов к группировке. Во-первых, можно группировать их по стереотипу. В таком случае получается один пакет с классами-сущностями, один с граничными классами, один с управляющими классами и т.д. Этот подход может быть полезен с точки зрения размещения готовой системы, поскольку все находящиеся на клиентских машинах пограничные классы уже оказываются в одном пакете.

Другой подход заключается в объединении классов по их функциональности. Например, в пакете Security (безопасность) содержатся все классы, отвечающие за безопасность приложения. В таком случае другие пакеты могут называться Employee Maintenance (Работа с сотрудниками), Reporting (Подготовка отчетов) и Error Handling (Обработка ошибок). Преимущество этого подхода заключается в возможности повторного использования.

Механизм пакетов применим к любым элементам модели, а не только к классам. Если для группировки классов не использовать некоторые эвристики, то она становится произвольной. Одна из них, которая в основном используется в UML, – это зависимость. Зависимость между

двумя пакетами существует в том случае, если между любыми двумя классами в пакетах существует любая зависимость. Таким образом, **диаграмма пакетов** (рис. 1.7) представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости являются элементами диаграммы классов, то есть диаграмма пакетов – это форма диаграммы классов.

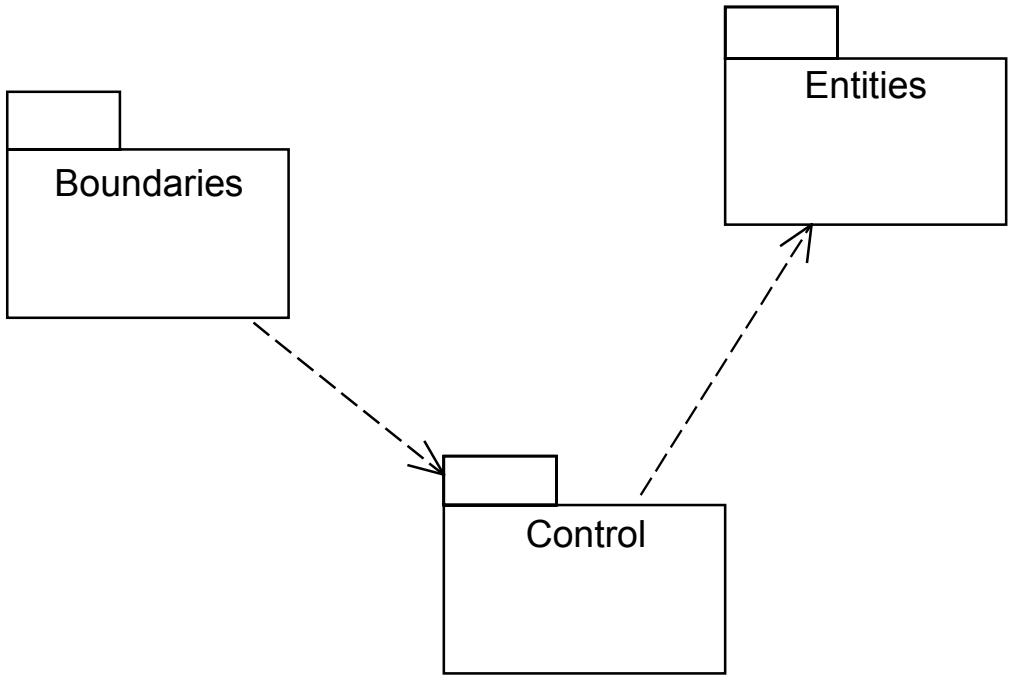

Рис. 1.7. Диаграмма пакетов

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Что касается классов, то причины для зависимостей могут быть самыми разными: один класс посылает сообщение другому; один класс включает часть данных другого класса; один класс использует другой в качестве параметра операции. Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может утратить свою силу.

Пакеты не дают ответа на вопрос, каким образом можно уменьшить количество зависимостей в вашей системе, однако они помогают выделить эти зависимости, а после того, как они все окажутся на виду, остается только поработать над снижением их количества. Диаграммы пакетов

можно считать основным средством управления общей структурой системы.

Пакеты являются жизненно необходимым средством для больших проектов. Их следует использовать в тех случаях, когда диаграмма классов, охватывающая всю систему в целом и размещенная на единственном листе бумаги формата А4, становится нечитаемой.

1.5.4. Атрибуты

Атрибут – это элемент информации, связанный с классом. Например, у класса Company (компания) могут быть атрибуты Name (Название), Address (Адрес) и NumberOfEmployees (Число служащих).

Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим может понадобиться указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута (attribute visibility).

У атрибута можно определить четыре возможных значения этого параметра. Рассмотрим каждый из них в контексте примера (рис. 1.8). Пусть у нас имеется класс Employee с атрибутом Address и класс Company:

- **Public (общий, открытый).** Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В таком случае класс Company может изменить значение атрибута Address класса Employee. В соответствии с нотацией UML общему атрибуту предшествует знак « + ».
- **Private (закрытый, секретный).** Соответствующий атрибут не виден никаким другим классом. Класс Employee будет знать значение атрибута Address и сможет изменять его, но класс Company не сможет его ни увидеть, ни редактировать. Если это понадобится, он должен попросить класс Employee просмотреть или изменить значение этого атрибута, что обычно делается с помощью общих операций. Закрытый атрибут обозначается знаком « – » в соответствии с нотацией UML.
- **Protected (защищенный).** Такой атрибут доступен только самому классу и его потомкам. Допустим, что у нас имеется два различных

типа сотрудников – с почасовой оплатой и на окладе. Таким образом, мы получаем два других класса HourlyEmp и SalariedEmp, являющихся потомками класса Employee. Защищенный атрибут Address можно просмотреть или изменить из классов Employee, HourlyEmp и SalariedEmp, но не из класса Company. Нотация UML для защищенного атрибута – это знак « # ».

– **Package or Implementation (пакетный).** Предполагает, что данный атрибут является общим, но только в пределах его пакета. Допустим, что атрибут Address имеет пакетную видимость. В таком случае он может быть изменен из класса Company, только если этот класс находится в том же пакете. Этот тип видимости не обозначается никаким специальным значком.

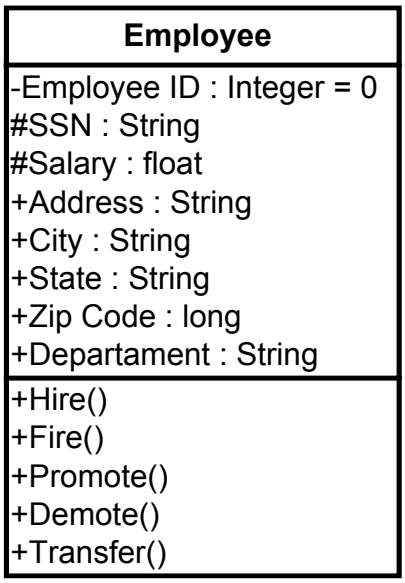

Рис. 1.8. Видимость атрибутов

В общем случае, атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код. С помощью закрытости или защищенности удается избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам этот атрибут. Задаваемые параметры видимости повлияют на генерируемый код.

1.5.5. Операции

Операции реализуют связанное с классом поведение. Операция включает три части – имя, параметры и тип возвращаемого значения. Параметры – это аргументы, получаемые операцией «на входе». Тип возвращаемого значения относится к результату действия операции.

На диаграмме классов можно показывать как имена операций, так и имена операций вместе с их параметрами и типом возвращаемого значения. Чтобы уменьшить загруженность диаграммы, полезно бывает на некоторых из них показывать только имена операций, а на других их полную сигнатуру.

В языке UML операции имеют следующую нотацию:

Имя Операции (аргумент1 : тип данных аргумента1, аргумент2 : тип данных аргумента2, ...) : тип возвращаемого значения

Следует рассмотреть четыре различных типа операций.

Операции реализации

Операции реализации (implementor operations) реализуют некоторые бизнес-функции. Такие операции можно найти, исследуя диаграммы взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциях, и каждое сообщение диаграммы, скорее всего, можно соотнести с операцией реализации.

Каждая операция реализации должна быть легко прослеживаема до соответствующего требования. Это достигается на различных этапах моделирования. Операция выводится из сообщения на диаграмме взаимодействия, сообщения исходят из подробного описания потока событий, который создается на основе варианта использования, а последний – на основе требований. Возможность проследить всю эту цепочку позволяет гарантировать, что каждое требование будет реализовано в коде, а каждый фрагмент кода реализует какое-то требование.

Операции управления

Операции управления (manager operations) управляют созданием и уничтожением объектов. В эту категорию попадают конструкторы и деструкторы классов.

Операции доступа

Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны просматривать или изменять их значения. Для этого существуют операции доступа (access operations).

Пусть, например, у нас имеется атрибут Salary класса Employee. Мы не хотим, чтобы все остальные классы могли изменять этот атрибут. Вместо этого к классу Employee мы добавляем две операции доступа – GetSalary и SetSalary. К первой из них, являющейся общей, могут обращаться и другие классы. Она просто получает значение атрибута Salary и возвращает его вызвавшему ее классу. Операция SetSalary также является общей, она помогает вызвавшему ее классу установить новое значение атрибута Salary. Эта операция может содержать любые правила и условия проверки, которые необходимо выполнить, чтобы зарплата могла быть изменена.

Такой подход дает возможность безопасно инкапсулировать атрибуты внутри класса, защитив их от других классов, но все же позволяет осуществить к ним контролируемый доступ. Создание операций Get и Set (получения и изменения значения) для каждого атрибута класса является стандартом.

Вспомогательные операции

Вспомогательными (helper operations) называются такие операции класса, которые необходимы ему для выполнения его ответственностей, но о которых другие классы не должны ничего знать. Это закрытые и защищенные операции класса.

Чтобы идентифицировать операции, выполните следующие действия:

1. Изучите диаграммы последовательности и кооперативные диаграммы. Большая часть сообщений на этих диаграммах является операциями реализации. Рефлексивные сообщения будут вспомогательными операциями.
2. Рассмотрите управляющие операции. Может потребоваться добавить конструкторы и деструкторы.
3. Рассмотрите операции доступа. Для каждого атрибута класса, с которым должны будут работать другие классы, надо создать операции Get и Set.

1.5.6. Связи

Связь представляет собой семантическую взаимосвязь между классами. Она дает классу возможность узнавать об атрибутах, операциях и связях другого класса. Иными словами, чтобы один класс мог послать сообщение другому на диаграмме последовательности или кооперативной диаграмме, между ними должна существовать связь.

Существуют четыре типа связей, которые могут быть установлены между классами: ассоциации, зависимости, агрегации и обобщения.

Ассоциации

Ассоциация (association) – это семантическая связь между классами. Их рисуют на диаграмме классов в виде обыкновенной линии.

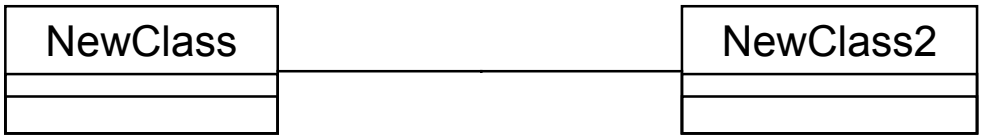

Рис. 1.9. Ассоциация

Ассоциации могут быть двунаправленными, как в примере, или однонаправленными. На языке UML двунаправленные ассоциации рисуют в виде простой линии без стрелок или со стрелками с обеих ее сторон. На однонаправленной ассоциации изображают только одну стрелку, показывающую ее направление.

Направление ассоциации можно определить, изучая диаграммы последовательности и кооперативные диаграммы. Если все сообщения на них отправляются только одним классом и принимаются только другим классом, но не наоборот, между этими классами имеет место однонаправленная связь. Если хотя бы одно сообщение отправляется в обратную сторону, ассоциация должна быть двунаправленной.

Ассоциации могут быть рефлексивными. Рефлексивная ассоциация предполагает, что один экземпляр класса взаимодействует с другими экземплярами этого же класса.

Зависимости

Связи зависимости (dependency) также отражают связь между классами, но они всегда однонаправлены и показывают, что один класс зависит от определений, сделанных в другом. Зависимости изображают в виде стрелки, проведенной пунктирной линией.

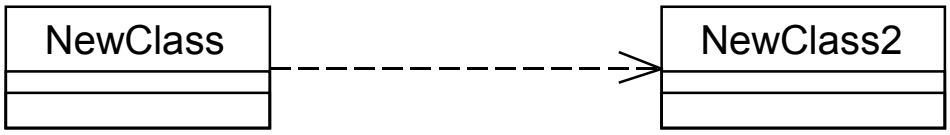

Рис. 1.10. Зависимость

При генерации кода для этих классов к ним не будут добавляться новые атрибуты. Однако, будут созданы специфические для языка операторы, необходимые для поддержки связи. Например, на языке C++ в код войдут необходимые операторы #include.

Агрегации

Агрегации (aggregations) представляют собой более тесную форму ассоциации. Агрегация – это связь между целым и его частью. Например, у вас может быть класс Автомобиль, а также классы Двигатель, Покрышки и классы для других частей автомобиля. В результате объект класса Автомобиль будет состоять из объекта класса Двигатель, четырех объектов Покрышек и т. д. Агрегации визуализируют в виде линии с ромбиком у класса, являющегося целым:

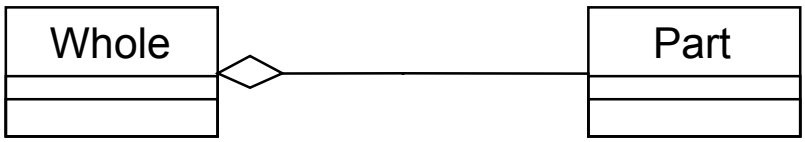

Рис. 1.11. Агрегация

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую композицией. Согласно

композиции, объект-часть может принадлежать только единственному целому, и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части.

Такое каскадное удаление нередко рассматривается как часть определения агрегации, однако оно всегда подразумевается в том случае, когда множественность роли составляет 1..1; например, если необходимо удалить Клиента, то это удаление должно распространиться и на Заказы (и, в свою очередь, на Строки заказа).

Обобщения

С помощью обобщений (generalization) показывают связи наследования между двумя классами. Большинство объектно-ориентированных языков непосредственно поддерживают концепцию наследования. Она позволяет одному классу наследовать все атрибуты, операции и связи другого. На языке UML связи наследования называют обобщениями и изображают в виде стрелок от класса-потомка к классу-предку:

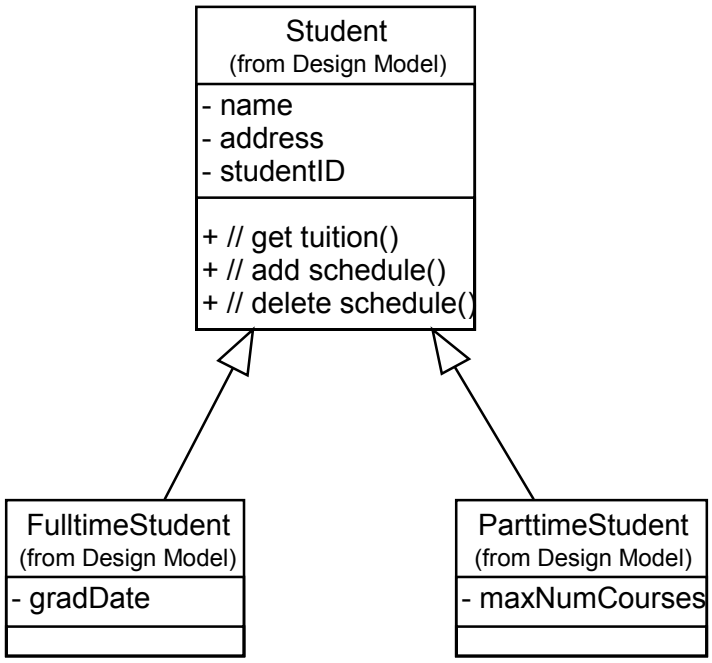

Рис. 1.12. Обобщение

Помимо наследуемых, каждый подкласс имеет свои собственные уникальные атрибуты, операции и связи.

Множественность

Множественность (multiplicity) показывает, сколько экземпляров одного класса взаимодействуют с помощью этой связи с одним экземпляром другого класса в данный момент времени.

Например, при разработке системы регистрации курсов в университете можно определить классы Course (курс) и Student (студент). Между ними установлена связь: у курсов могут быть студенты, а у студентов – курсы. Вопросы, на который должен ответить параметр множественности: «Сколько курсов студент может посещать в данный момент? Сколько студентов может за раз посещать один курс?»

Так как множественность дает ответ на оба эти вопроса, её индикаторы устанавливаются на обоих концах линии связи. В примере регистрации курсов мы решили, что один студент может посещать от нуля до четырех курсов, а один курс могут слушать от 10 до 20 студентов. На диаграмме классов это можно изобразить, как показано на рис. 1.13.

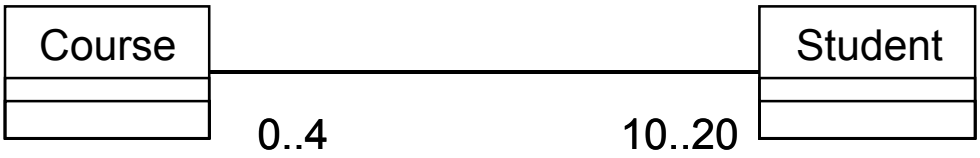

Рис. 1.13. Множественность

В языке UML приняты следующие нотации для обозначения множественности:

Множественность	**Значение**
0..*	Ноль или больше
1..*	Один или больше
0..1	Ноль или один
1..1 (сокращенная запись: 1)	Ровно один

Имена связей

Связи можно уточнить с помощью имен связей или ролевых имен. Имя связи – это обычно глагол или глагольная фраза, описывающая, зачем она нужна. Например, между классом Person (человек) и классом Company (компания) может существовать ассоциация. Можно задать в связи с этим вопрос, является ли объект класса Person клиентом компании, её сотрудником или владельцем? Чтобы определить это, ассоциацию можно назвать «employs» (нанимает):

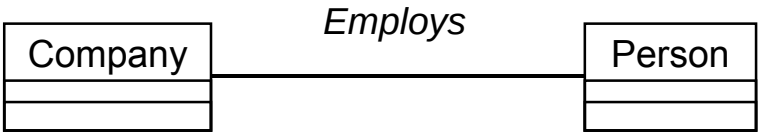

Рис. 1.14. Имя связи

Имена у связей определять не обязательно. Обычно это делают, если причина создания связи не очевидна. Имя показывают около линии соответствующей связи.

Роли

Ролевые имена применяют в связях ассоциации или агрегации вместо имен для описания того, зачем эти связи нужны. Возвращаясь к примеру с классами Person и Company, можно сказать, что класс Person играет роль сотрудника класса Company. Ролевые имена – это обычно имена существительные или основанные на них фразы, их показывают на диаграмме рядом с классом, играющим соответствующую роль. Как правило, пользуются или ролевым именем, или именем связи, но не обоими сразу. Как и имена связей, ролевые имена не обязательны, их дают, только если цель связи не очевидна. Пример ролей приводится ниже:

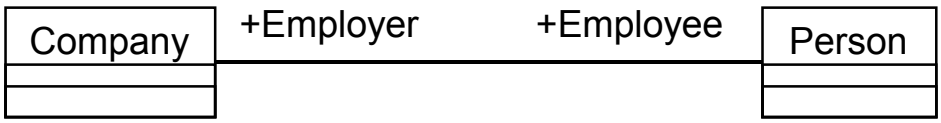

Рис. 1.15. Ролевые имена

1.6. Диаграммы состояний

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий. Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой. Наиболее распространенная форма, используемая в объектно-ориентированных методах, впервые применялась в методе OMT и впоследствии была адаптирована Гради Бучем.

На рис. 1.16 приводится пример диаграммы состояний для банковского счета. Из данной диаграммы видно, в каких состояниях может существовать счет. Можно также видеть процесс перехода счета из одного состояния в другое. Например, если клиент требует закрыть открытый счет, он переходит в состояние «Закрыт». Требование клиента называется событием (event), именно такие события и вызывают переход из одного состояния в другое.

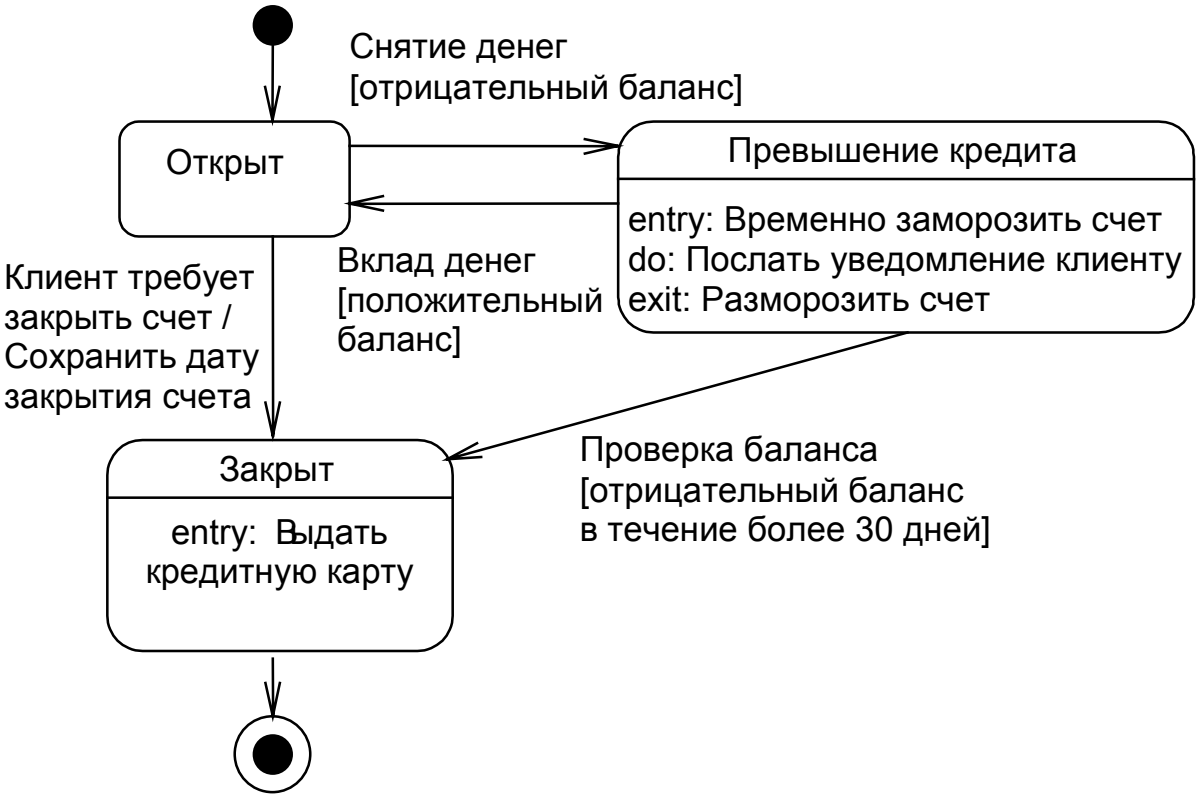

Рис. 1.16. Диаграмма состояний для класса Account

Если клиент снимает деньги с открытого счета, он может перейти в состояние «Превышение кредита». Это происходит, только если баланс по этому счету меньше ноля, что отражено условием [отрицательный баланс] на нашей диаграмме. Заключенное в квадратных скобках условие (guard condition) определяет, когда может или не может произойти переход из одного состояния в другое.

На диаграмме имеются два специальных состояния – начальное (start) и конечное (stop). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время, может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы. В нашем примере при превышении кредита клиенту посылается соответствующее сообщение. Процессы, происходящие, когда объект находится в определенном состоянии, называются действиями (actions).

С состоянием можно связывать данные пяти типов: деятельность, входное действие, выходное действие, событие и история состояния. Рассмотрим каждый из них в контексте диаграммы состояний для класса Account системы ATM.

Деятельность

Деятельностью (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Например, когда счет находится в состоянии «Закрыт», происходит возврат кредитной карточки пользователю. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие.

Входное действие

Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. В примере счета в банке, когда он переходит в состояние «Превышен счет», выполняется действие «Временно заморозить счет», независимо от того, откуда объект перешел в это состояние. Таким образом, данное действие осуществляется не после того, как объект перешел в это состояние, а, скорее, как часть этого перехода. В отличие от деятельности, входное действие рассматривается как непрерываемое.

Входное действие также показывают внутри состояния, ему предшествует слово entry (вход) и двоеточие.

Выходное действие

Выходное действие (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. В нашем примере при выходе объекта Account из состояния «Превышен счет», независимо от того, куда он переходит, выполняется действие «Разморозить счет». Оно является частью процесса такого перехода. Как и входное, выходное действие является непрерываемым.

Выходное действие изображают внутри состояния, ему предшествует слово exit (выход) и двоеточие.

Поведение объекта во время деятельности, при входных и выходных действиях может включать отправку события другому объекту. Например, объект account (счет) может посылать событие объекту card reader (устройство чтения карты). В этом случае описанию деятельности, входного действия или выходного действия предшествует знак « ^ ». Соответствующая строка на диаграмме выглядит как

Do: ^Цель.Событие(Аргументы)

Здесь Цель – это объект, получающий событие, Событие – это посылаемое сообщение, а Аргументы являются параметрами посылаемого сообщения.

Деятельность может также выполняться в результате получения объектом некоторого события. Например, объект account может быть

в состоянии Открыто. При получении некоторого события выполняется определенная деятельность.

Переходом (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим.

Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии.

У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события. Рассмотрим каждое из них в контексте примера ATM.

События

Событие (event) – это то, что вызывает переход из одного состояния в другое. В нашем примере событие «Клиент требует закрыть» вызывает переход счета из открытого в закрытое состояние. Событие размещают на диаграмме вдоль линии перехода.

На диаграмме для отображения события можно использовать как имя операции, так и обычную фразу. В нашем примере события описаны обычными фразами. Если вы хотите использовать операции, то событие «Клиент требует закрыть» можно было бы назвать RequestClosure().

У событий могут быть аргументы. Так, событие «Сделать вклад», вызывающее переход счета из состояния «Превышен счет» в состояние «Открыт», может иметь аргумент Amount (Количество), описывающий сумму депозита.

Большинство переходов должны иметь события, так как именно они, прежде всего, заставляют переход осуществиться. Тем не менее, бывают и автоматические переходы, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, позволяющей осуществиться входным действиям, деятельности и выходным действиям.

Ограждающие условия

Ограждающие условия (guard conditions) определяют, когда переход может, а когда не может осуществиться. В нашем примере событие «Сделать вклад» переведет счет из состояния «Превышение счета» в состояние «Открыт», но только если баланс будет больше нуля. В противном случае переход не осуществится.

Ограждающие условия изображают на диаграмме вдоль линии перехода после имени события, заключая их в квадратные скобки.

Ограждающие условия задавать необязательно. Однако если существует несколько автоматических переходов из состояния, необходимо определить для них взаимно исключающие ограждающие условия. Это поможет читателю диаграммы понять, какой путь перехода будет автоматически выбран.

Действие

Действием (action), как уже говорилось, является непрерываемое поведение, осуществляющееся как часть перехода. Входные и выходные действия показывают внутри состояний, поскольку они определяют, что происходит, когда объект входит или выходит из него. Большую часть действий, однако, изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния.

Например, при переходе счета из открытого в закрытое состояние выполняется действие «Сохранить дату закрытия счета». Это непрерываемое поведение осуществляется только во время перехода из состояния «Открыт» в состояние «Закрыт».

Действие рисуют вдоль линии перехода после имени события, ему предшествует косая черта.

Событие или действие могут быть поведением внутри объекта, а могут представлять собой сообщение, посылаемое другому объекту. Если событие или действие посылается другому объекту, перед ним на диаграмме помещают знак « ^ ».

Диаграммы состояний не надо создавать для каждого класса, они применяются только в сложных случаях. Если объект класса может

существовать в нескольких состояниях и в каждом из них ведет себя по-разному, для него может потребоваться такая диаграмма.

1.7. Диаграммы деятельности

В отличие от большинства других средств UML, диаграммы деятельностей не имеют явно выраженного источника в предыдущих работах Буча, Рамбо и Якобсона, и заимствуют идеи из нескольких различных методов, в частности, метода моделирования состояний SDL и сетей Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов [Фаулер-1999].

Подобно большинству других средств, моделирующих поведение, диаграммы деятельностей обладают определенными достоинствами и недостатками, поэтому их лучше всего использовать в сочетании с другими средствами.

Самым большим достоинством диаграмм деятельностей является поддержка параллелизма. Благодаря этому они являются мощным средством моделирования потоков работ и, по существу, параллельного программирования. Самый большой их недостаток заключается в том, что связи между действиями и объектами просматриваются не слишком четко.

Эти связи можно попытаться определить, используя для деятельностей метки с именами объектов, но этот способ не обладает такой же простой непосредственностью, как у диаграмм взаимодействия. Диаграммы деятельностей предпочтительнее использовать в следующих ситуациях:

- Анализ варианта использования. На этой стадии нас не интересует связь между действиями и объектами, а нужно только понять, какие действия должны иметь место и каковы зависимости в поведении системы. Связывание методов и объектов выполняется позднее с помощью диаграмм взаимодействия.
- Анализ потоков работ (workflow) в различных вариантах использования. Когда варианты использования взаимодействуют друг с другом, диаграммы деятельностей являются мощным средством представления и анализа их поведения.

1.8. Диаграммы компонентов

Диаграммы компонентов показывают, как выглядит модель на физическом уровне. На них изображены компоненты программного обеспечения и связи между ними. При этом на такой диаграмме выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

На рис. 1.17 изображена одна из диаграмм компонентов для системы ATM.

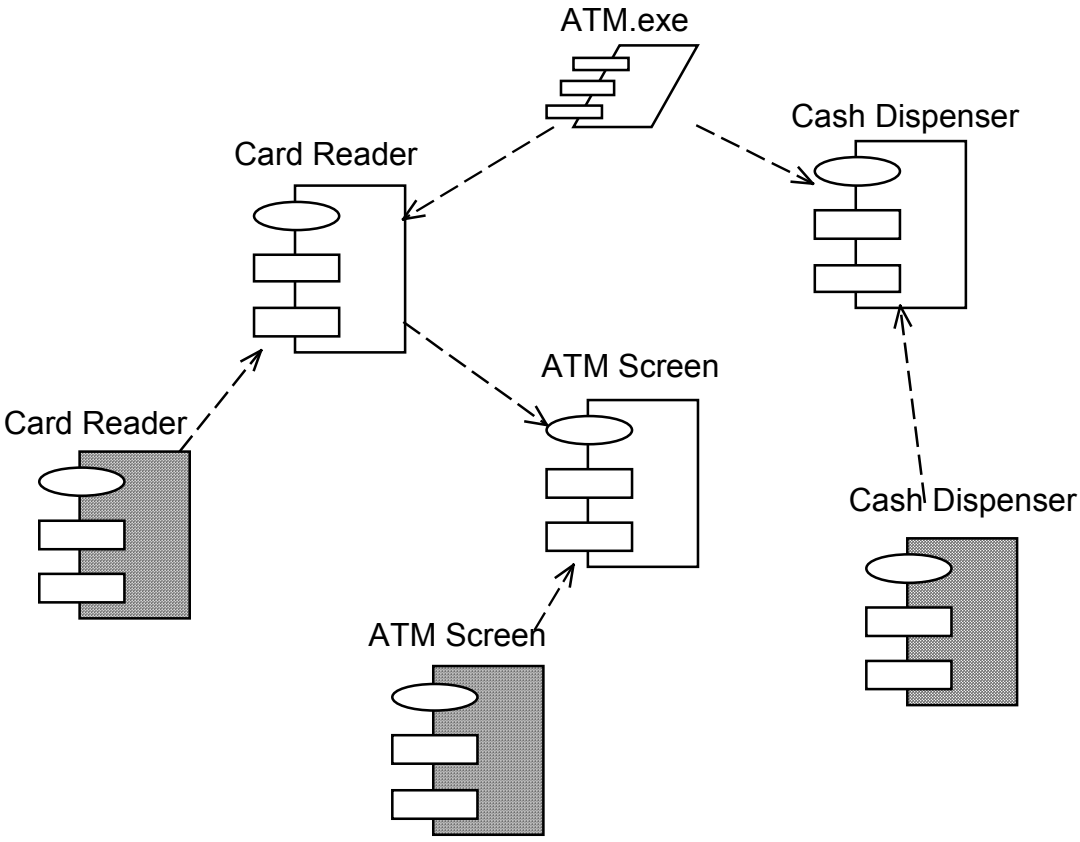

Рис. 1.17. Диаграмма компонентов для клиента ATM

На этой диаграмме показаны компоненты клиента системы ATM. В данном случае система разрабатывается на языке C++. У каждого класса

имеется свой собственный заголовочный файл и файл с расширением .CPP, так что каждый класс преобразуется в свои собственные компоненты на диаграмме. Например, класс ATM screen преобразуется в компонент ATM Screen диаграммы. Он преобразуется также и во второй компонент ATM Screen. Вместе эти два компонента представляют тело и заголовок класса ATM Screen. Выделенный темным компонент называется спецификацией пакета (package specification) и соответствует файлу тела класса ATM Screen на языке C++ (файл с расширением .CPP). Невыделенный компонент также называется спецификацией пакета, но соответствует заголовочному файлу класса языка C++ (файл с расширением .H). Компонент ATM.exe является спецификацией задачи и представляет поток обработки информации (thread of processing). В данном случае поток обработки является исполняемой программой.

Компоненты соединены штриховой линией, что соответствует зависимостям между ними. Например, класс Card Reader зависит от класса ATM Screen. Это означает, что, для того, чтобы класс Card Reader мог быть скомпилирован, класс ATM Screen должен уже существовать. После компиляции всех классов может быть создан исполняемый файл ATMClient.exe.

Пример ATM содержит два потока обработки и, таким образом, получаются два исполняемых файла. Один из них – это клиент ATM, он содержит компоненты Cash Dispenser, Card Reader и ATM Screen. Второй файл – это сервер ATM, включающий в себя компонент Account. Диаграмма компонентов для сервера ATM показана на рис. 1.18.

Как видно из примера, у системы может быть несколько диаграмм компонентов, в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема является пакетом компонентов. В общем случае, пакеты – это совокупности компонентов. Пример ATM содержит два пакета: клиент ATM и сервер ATM.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие

классов реализованным компонентам. Она нужна там, где начинается генерация кода.

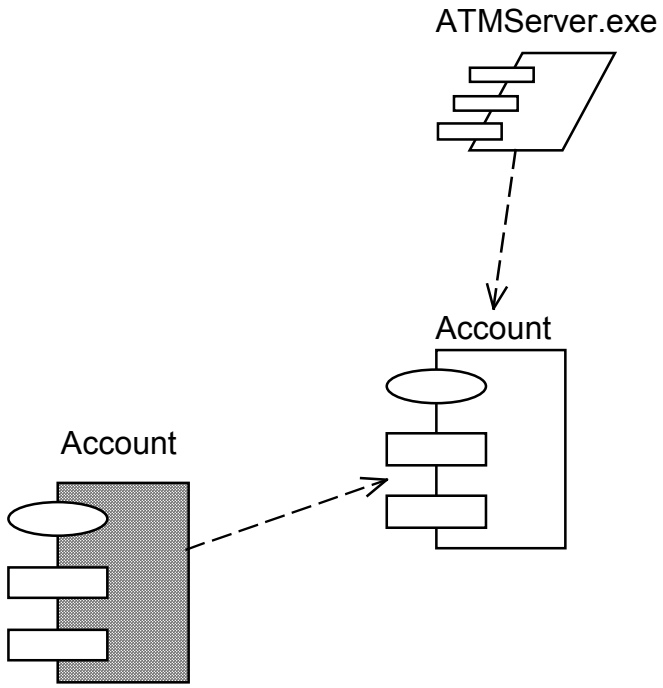

Рис. 1.18. Диаграмма Компонентов для сервера ATM

1.9. Диаграммы размещения

Диаграмма размещения (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать маршруты перемещения объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства – в большинстве случаев, часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мэйнфреймом.

Диаграмма размещения показывает физическое расположение сети и местонахождение в ней различных компонентов. В нашем примере система ATM состоит из большого количества подсистем, выполняемых на отдельных физических устройствах, или узлах (node). Диаграмма размещения для системы ATM показана на рис. 1.19.

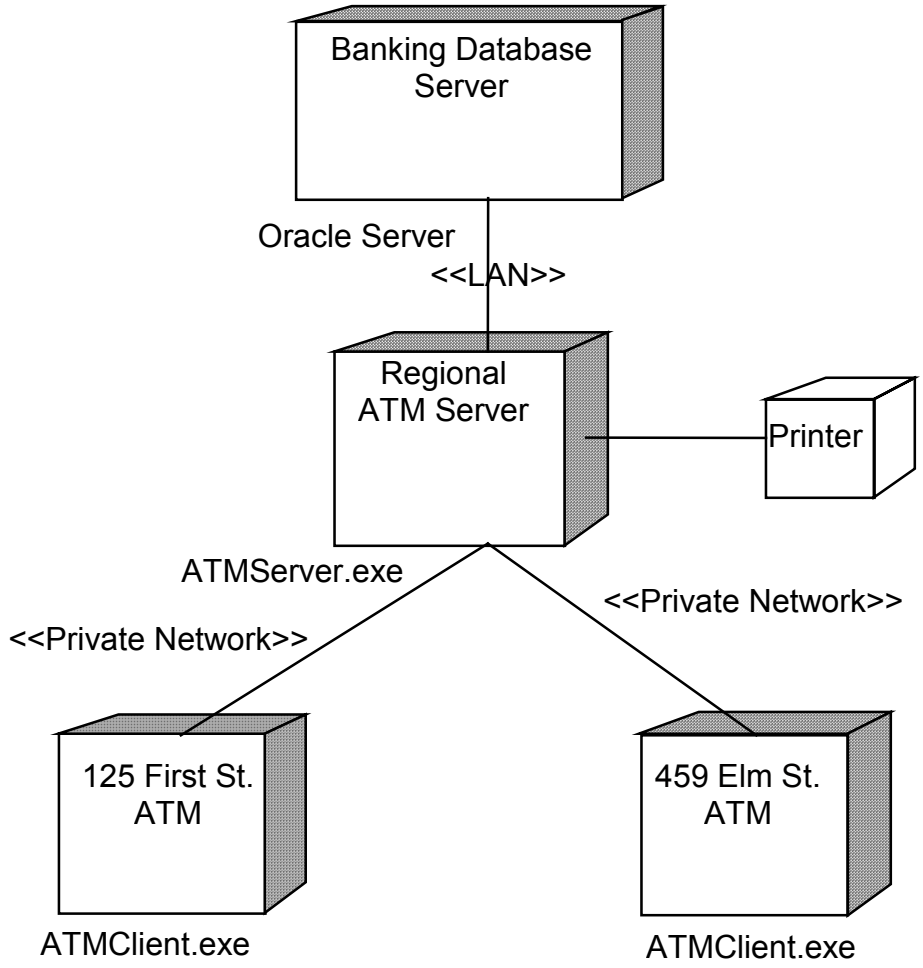

Рис. 1.19. Диаграмма размещения для системы ATM

Из данной диаграммы можно узнать о физическом размещении системы. Клиентские программы ATM будут работать в нескольких местах на различных сайтах. Через закрытые сети будет осуществляться их сообщение с региональным сервером ATM. На нём будет работать программное обеспечение сервера ATM. В свою очередь, посредством локальной сети региональный сервер будет сообщаться с сервером банковской базы данных, работающим под управлением Oracle. Наконец, с региональным сервером ATM соединен принтер.

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом, чтобы понять физическое размещение системы и расположение её отдельных подсистем.