2. Введение в объектно-ориентированный анализ и проектирование

2.1. OOP, OOD и OOA

Объектно-ориентированное программирование (object-oriented programming, OOP) - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

1) OOP использует в качестве базовых элементов объекты, а не алгоритмы;
2) каждый объект является экземпляром какого-либо определенного класса;
3) классы организованы иерархически.

Программа будет объектно-ориентированной только при условии соблюдения всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOP, а называется программированием на основе абстрактных типов данных.

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

1) поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций, и собственные данные, с ограничением доступа к ним;

2) объекты относятся к соответствующим типам (классам);

3) классы могут наследовать атрибуты суперклассов.

Поддержка наследования в таких языках означает возможность установления отношения «is-a» («есть», «это есть», «это»), например, красная роза - это цветок, а цветок - это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным, такие языки называют объектными. Согласно этому определению объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, а Ada - объектный язык. Но поскольку объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы объектно-ориентированного проектирования.

Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование основное внимание уделяет правильному и эффективному структурированию сложных систем.

Объектно-ориентированное проектирование (object-oriented design, OOD) - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части:

1) объектно-ориентированное проектирование основывается на объектно-ориентированной декомпозиции;

2) объектно-ориентированное проектирование использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором - алгоритмами.

Объектно-ориентированный анализ направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ (object-oriented analysis, OOA) - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся OOA, OOD и OOP? На результатах OOA формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP.

2.2. Сложность, присущая программному обеспечению

2.2.1. Простые и сложные программные системы

Звезда в преддверии коллапса; ребенок, который учится читать; клетки крови, атакующие вирус, - это только некоторые из потрясающе сложных объектов физического мира. Компьютерные программы тоже бывают сложными, однако их сложность совершенно другого рода.

Конечно, не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Это не означает, что такие системы плохо сделаны или, что их создатели имеют низкую квалификацию. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заме-

нить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна.

Мы будем рассматривать процесс разработки сложных программных комплексов. Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые управляют или сами управляются событиями физического мира, и для которых ресурсы времени и памяти ограничены; задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами; системы управления и контроля за реальными процессами (например, диспетчеризация воздушного или железнодорожного транспорта). Системы подобного типа обычно имеют длительное время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. Среди промышленных программ также встречаются среды разработки, которые упрощают создание приложений в конкретных областях и программы, которые имитируют определенные стороны человеческого интеллекта.

Существенная черта промышленной программы - уровень сложности. Один разработчик практически не в состоянии охватить все аспекты такой системы. Сложность здесь неизбежна. С ней можно справиться, но избавиться от нее нельзя.

Конечно, среди разработчиков есть гении, которые в одиночку могут выполнить работу группы обычных людей и добиться в своей области успеха, сравнимого с достижениями Франка Ллойда Райта или Леонардо да Винчи. Такие люди нужны как архитекторы, которые изобретают новые идиомы, механизмы и основные идеи, используемые затем при разработке других систем. Однако, как замечает Петерс: «В мире очень мало гениев, и не надо думать, будто в среде программистов их доля выше средней» [3]. Несмотря на то, что все люди чуточку гениальны, в промышленном программировании не приходится полагаться только на гениальность. Поэтому нужно рассмотреть более надежные способы конструирования сложных систем. Но сначала ответим на вопрос: почему сложность присуща всем большим программным системам?

2.2.2. Почему программному обеспечению присуща сложность?

Брукс отмечает, что сложность программного обеспечения - отнюдь не случайное его свойство [3]. Сложность вызывается четырьмя основными причинами:

1) сложностью реальной предметной области, из которой исходит заказ на разработку;
2) трудностью управления процессом разработки;

3) необходимостью обеспечения достаточной гибкости программы;
4) неудовлетворительными способами описания поведения больших дискретных систем.

Подробнее остановимся на каждой из причин.

1. ***Сложность реального мира***. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Даже в общих чертах, достаточно трудно понять, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность, стоимость, выживаемость и надежность. Сложность задачи и порождает ту сложность программного продукта, о которой пишет Брукс.

Эта внешняя сложность обычно возникает из-за «нестыковки» между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, «разбавленных» немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов - схем, прототипов - и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Большая программная система - это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Большие системы должны эволюционировать в

процессе их использования. Встает задача о сопровождении программного обеспечения.

2. ***Трудности управления процессом разработки.*** Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы отнюдь не входит в число ее главных достоинств, поэтому нужно делать исходные тексты более компактными. Однако новые требования для каждой новой системы неизбежны, а они приводят к необходимости либо создавать много программ «с нуля», либо пытаться по-новому использовать существующие. Всего 20 лет назад программы объемом в несколько тысяч строк на ассемблере выходили за пределы наших возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

3. ***Гибкость программного обеспечения.*** Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления стальных балок под будущее здание. Однако в программной индустрии такая практика - дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким делом.

4. ***Проблема описания поведения больших дискретных систем.*** Когда мы кидаем вверх мяч, мы можем достоверно предсказать его траекторию, потому что знаем, что в нормальных условиях здесь действуют известные физические законы. Мы бы очень удивились, если бы, кинув мяч с

чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и резко изменил направление движения. В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Д. Парнас [3] пишет: «когда мы говорим, что система описывается непрерывной функцией, мы имеем ввиду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных». С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовала на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пике. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

Наряду с приведенными причинами сложности, работу со сложными системами усугубляют ограниченные физические возможности человека, которые связаны с объемом его краткосрочной памяти. Психологические

исследования показали, что количество структурных единиц, за которыми одновременно может следить человек, равно 7±2.

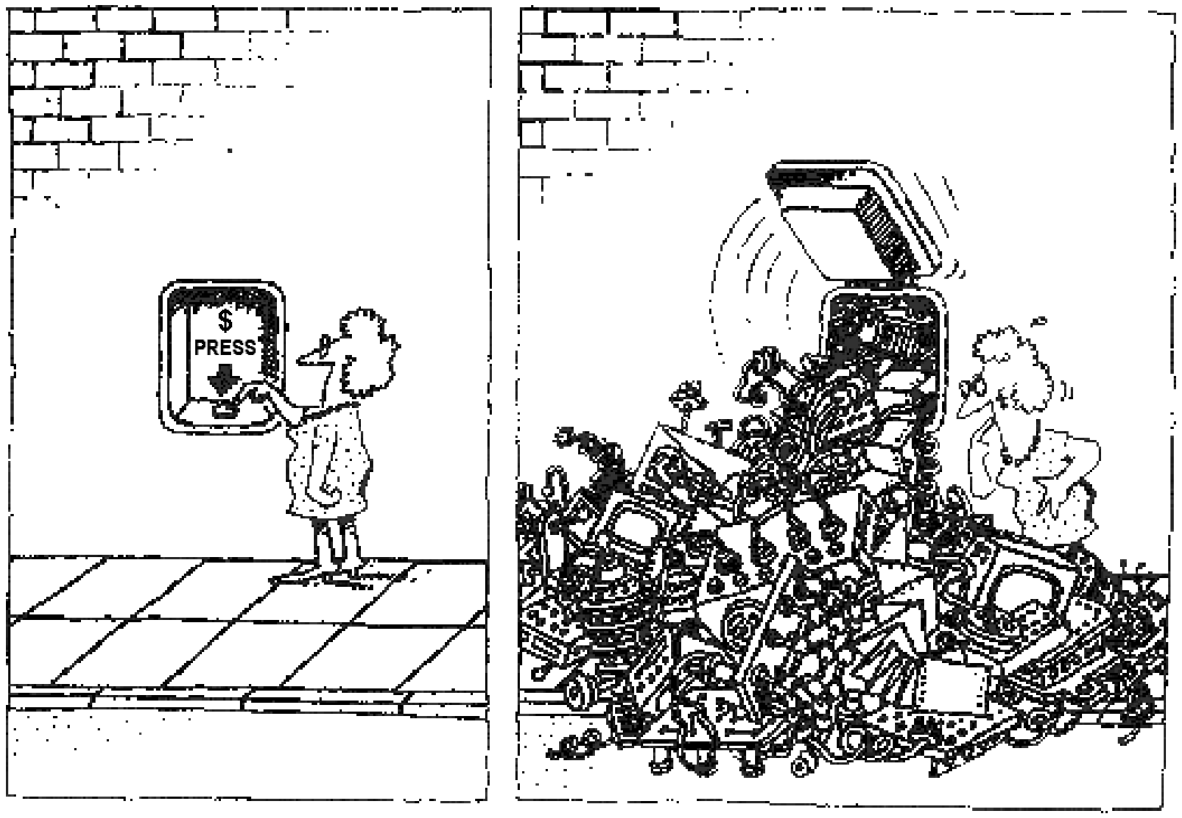


Рис. 2.1. Задача разработчиков программной системы – создать иллюзию простоты

Как же справляться со сложными системами? Можно заметить, что в других областях, например, архитектуре или управлении так же присутствуют сложные системы. Их можно встретить повсюду, и они успешно функционируют. Значит со сложностью можно справиться. Рассмотрим способы преодоления сложности.

2.2.3. Пять признаков сложной системы

Исходя из изучения структуры сложных систем, выводят пять общих признаков любой сложной системы.

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы. Это связано, скорее всего, с особенностью нашего восприятия - мы можем понять лишь те системы, которые имеют иерархическую структуру.

2. Выбор того, какие компоненты в данной системе считаются элементарными, относительно произволен и зависит от исследователя. Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого. Одна и та же роль элемента в системе может быть важной для одного и незначительной для другого исследователя.

3. Внутрикомпонентная связь в системе обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами. Это дает возможность изолированно изучать каждую часть.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных. Подобно тому, как в растении много разных клеток, но тип элементарного элемента один – клетка.

5. Любая работающая сложная система является результатом развития работавшей ранее более простой системы. Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматриваемые как сложные, делятся на элементарные, и из них строятся более сложные системы. Не возможно сразу правильно создать элементарные объекты. С ними нужно вначале поработать, чтобы больше узнать о реальном поведении системы, а затем уже совершенствовать.

2.2.4. Декомпозиция, абстракция и иерархия

Проблема ограниченности восприятия человеком сложных систем решается с помощью декомпозиции, абстракции и иерархии.

1. ***Иерархия. Каноническая форма сложной системы.*** В любой сложной системе можно выделить два типа иерархий. Первый тип – это ***иерархия «обобщение-специализация»*** (другое название «общее и частное» или «is-a»), в которой подкласс представляет собой специализированный частный случай своего суперкласса. Эта иерархия по-другому еще называется структура классов. Например, роза – это цветок, цветок – это растение. Второй тип иерархии – это ***иерархия «быть частью»*** (другое название «целое-часть» или «part-of»), когда про элемент говорят, что он часть другого. По-другому такая иерархия называется структура объектов. Например, лепестки - часть цветов.

Внесение иерархии позволяет четко классифицировать объекты и их свойства, а, значит, помогает справиться с присущей им сложностью.

Понятия структура классов и структура объектов, объединенные с пятью признаками сложных систем, называются ***канонической формой***

сложной системы (рис. 2.2). Те программные системы, в которых хорошо продумана каноническая форма, являются наиболее успешными.

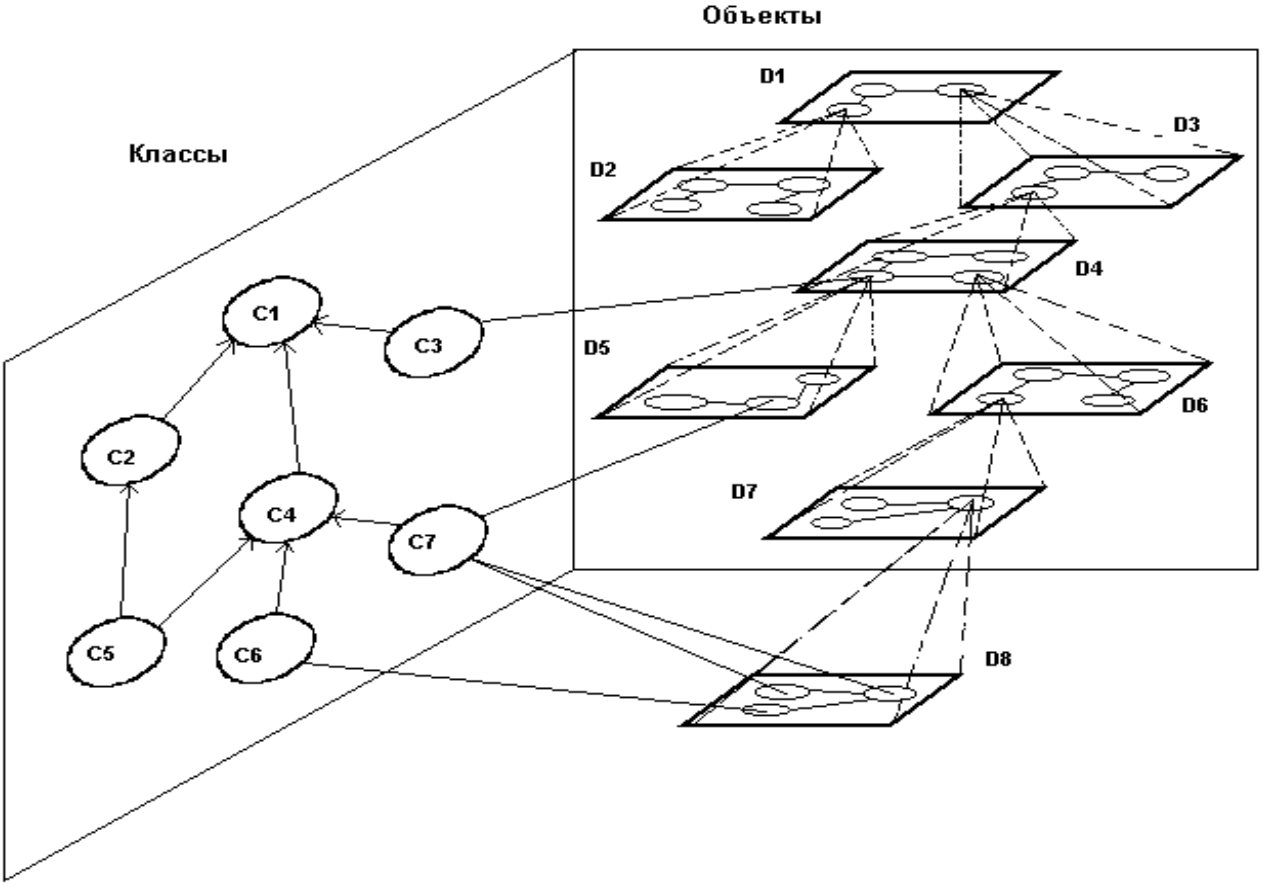


Рис. 2.2. Каноническая форма сложной системы

2. ***Декомпозиция.*** Способ управления сложными системами был известен давно, это принцип: «divide et impera» («разделяй и властвуй»). Программную систему можно разделить на подсистемы и совершенствовать независимо, при этом в уме не нужно держать информацию обо всей системе сразу.

Различают два типа декомпозиции - алгоритмическая и объектно-ориентированная. При ***алгоритмической декомпозиции*** основное внимание концентрируется на порядке происходящих событий (процессах). Примером алгоритмической декомпозиции может являться структурная схема, которая показывает связи между функциональными элементами системы (рис. 2.3). ***Объектно-ориентированная декомпозиция*** придает основное значение объектам, которые в процессе взаимодействия друг с другом определяют поведение системы. Объектно-ориентированная декомпозиция основана на объектах, а не на алгоритмах (рис. 2.4). Обе декомпозиции важны. Но начинать лучше с объектно-ориентированной.

3. ***Абстракция*** – это выделение существенных, с точки зрения наблюдателя, особенностей объекта и игнорирование несущественных.

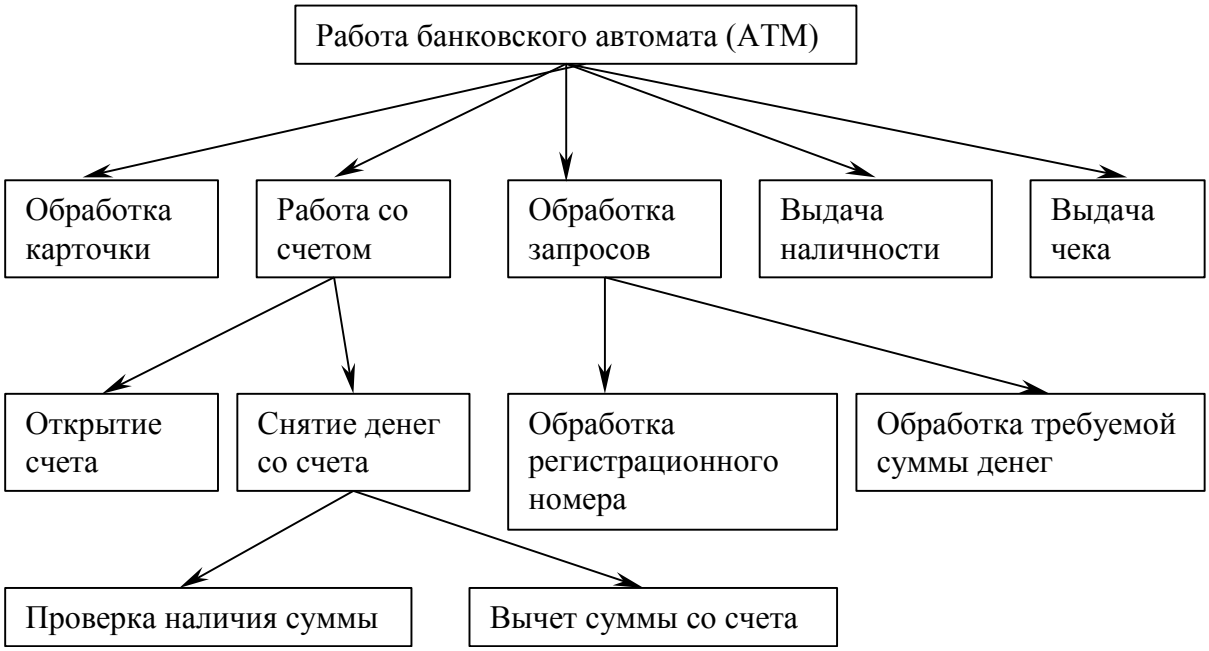


Рис. 2.3. Пример алгоритмической декомпозиции

Рис. 2.4. Пример объектно-ориентированной декомпозиции

2.3. Основные принципы объектной модели

Объектно-ориентированная технология основывается на объектной модели (ОМ). Ее основными принципами являются: абстрагирование, инкапсуляция, модульность, иерархия, типизация, параллелизм, сохраняемость. Первые четыре элемента являются главными, потому что без любого из них модель не будет объектно-ориентированной. Все принципы не новы, но в объектно-ориентированной модели они впервые применены в совокупности. Подробнее рассмотрим каждый из перечисленных принципов.

2.3.1. Абстрагирование

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

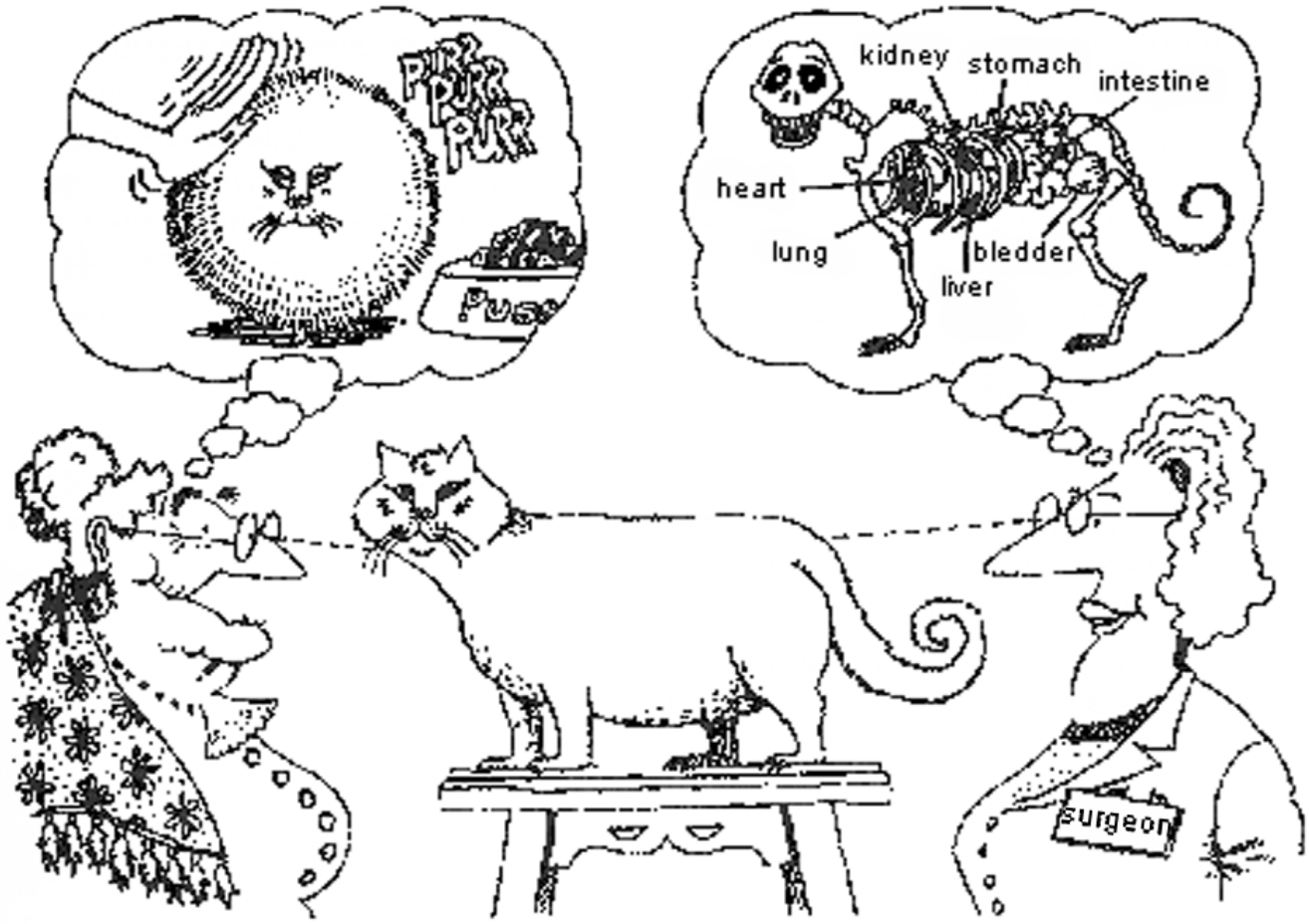


Рис. 2.5. Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта

2.3.2. Инкапсуляция

Абстракция и инкапсуляция дополняют друг друга. Абстрагирование направлено на (наблюдаемое) поведение объектов, а инкапсуляция занимается (внутренним) устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрывается и внутренняя структура объекта, и реализация его методов. Таким образом, инкапсуляция скрывает детали реализации объекта.

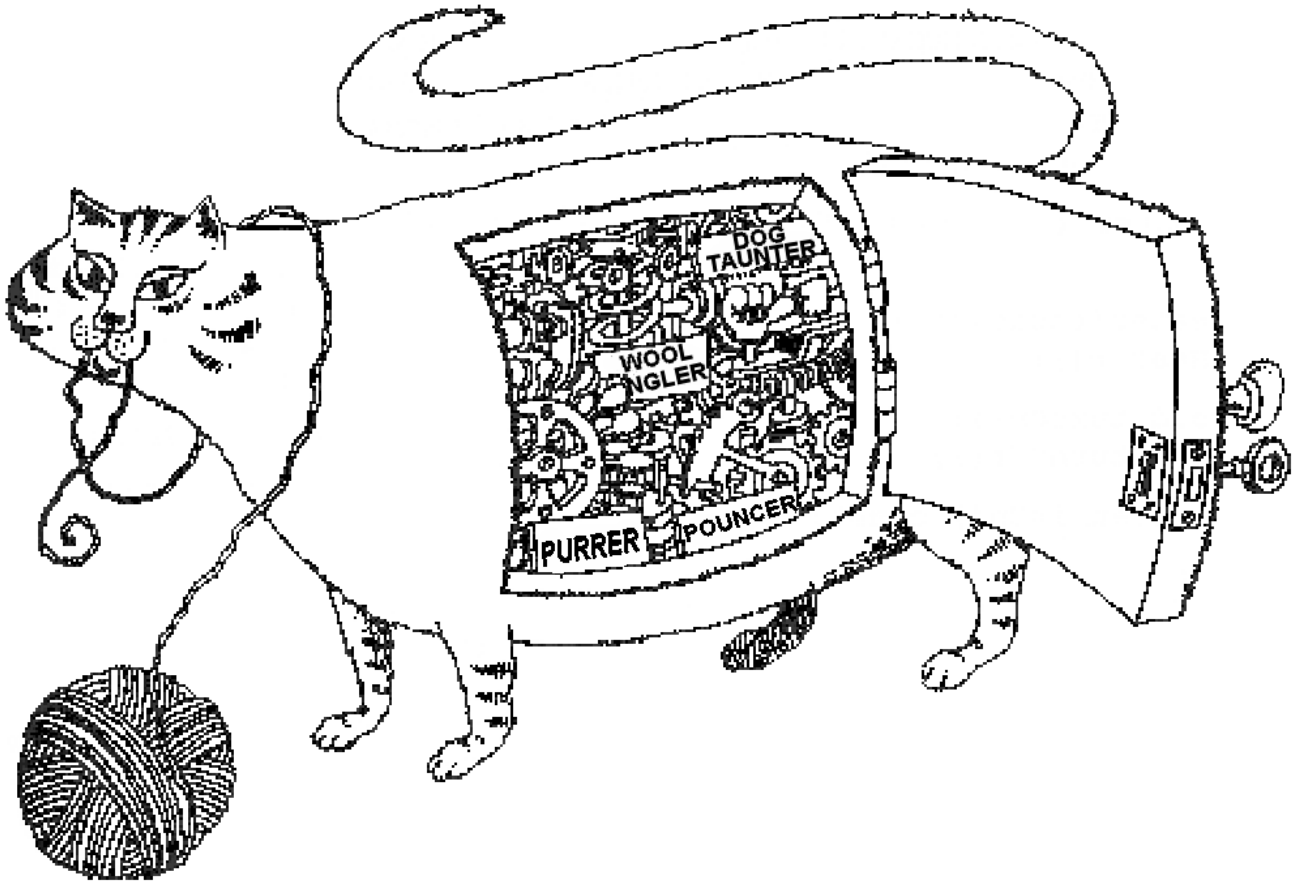


Рис. 2.6. Инкапсуляция скрывает детали реализации объекта

Практически совместная работа абстракции и инкапсуляции означает, что в классе существует две части - интерфейс и реализация. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. В интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от

других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

2.3.3. Модульность

В традиционном структурном проектировании под ***модульным принципом*** понимается разделение программы на части (модули). Для уменьшения сложности программы стремятся чтобы модули были небольшими и в высокой степени независимыми. Независимость модулей достигается двумя методами оптимизации: усилением связей внутри модуля и ослаблением взаимосвязи между модулями. Таким образом, системы должны иметь внутренние связанные, но слабо сцепленные между собой модули.

В объектно-ориентированном проектировании модульность – это искусство физически разделять классы и объекты, составляющие логическую структуру проекта. Следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями.

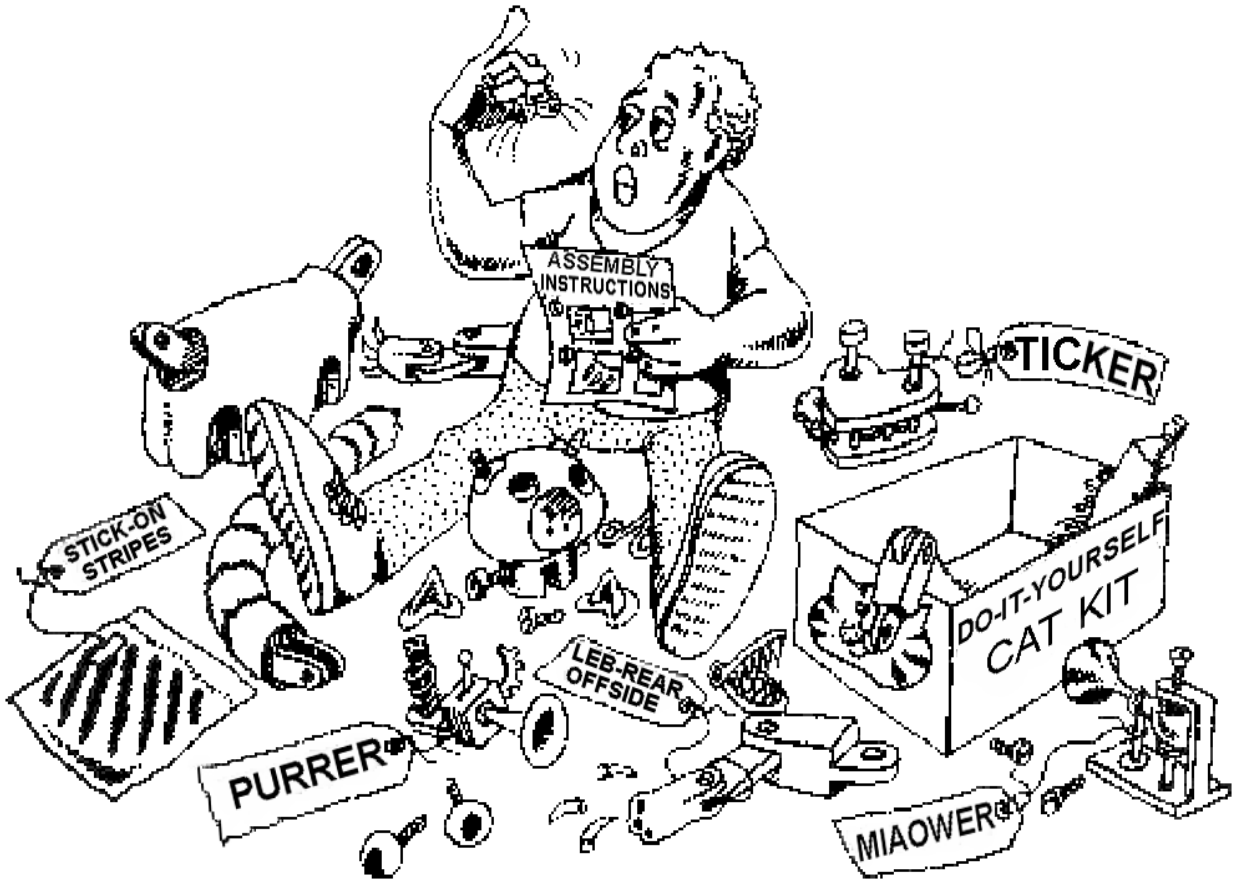


Рис. 2.7. Модульность позволяет хранить абстракции раздельно

Принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

2.3.4. Иерархия

Абстракция - полезная вещь. Однако часто число абстракций в системе намного превышает предел, доступный нашим умственным возможностям. Инкапсуляция позволяет до какой-то степени устранить этот недостаток, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Однако этого недостаточно. Значительное упрощение понимания сложной задачи достигается за счет образования из абстракций иерархической структуры.

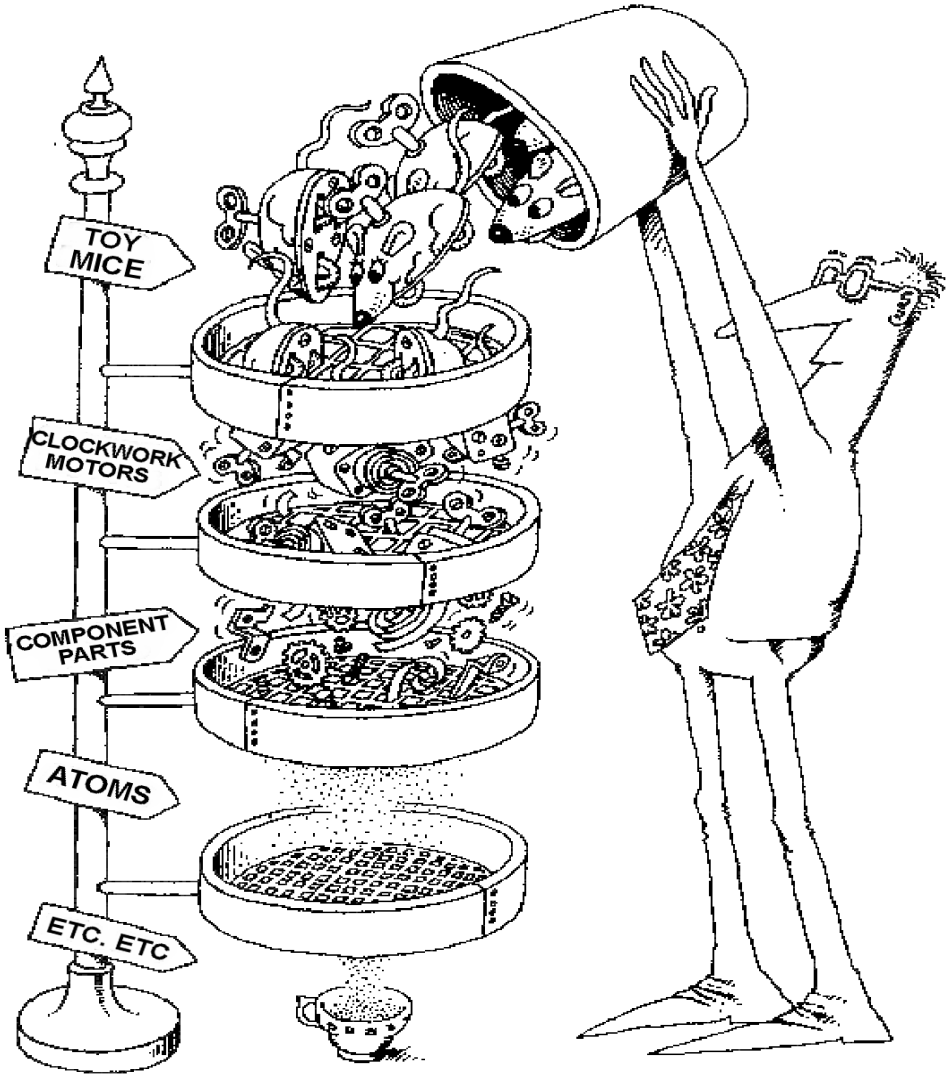


Рис. 2.8. Абстракции образуют иерархию

Иерархия – это упорядочение абстракций, разложение их по уровням. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия «is-a») и структура объектов (иерархия «part of»). С иерархией в объектно-ориентированной парадигме связаны такие понятия как наследование, полиморфизм и агрегация, подробнее они рассматриваются разделе «Классы».

2.3.5. Типизация

Типизация – это способ исключить использование объектов одного класса вместо другого или управление таким использованием. Понятие типа взято из теории абстрактных типов данных. Идея согласования типов занимает в типизации центральное место. Примером типизации может служить использование физических единиц измерения. Деля расстояние на время, мы ожидаем получить скорость, а не вес. Умножение температуры на силу не имеет смысла, а умножение расстояния на силу имеет смысл. Все это примеры типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракции.

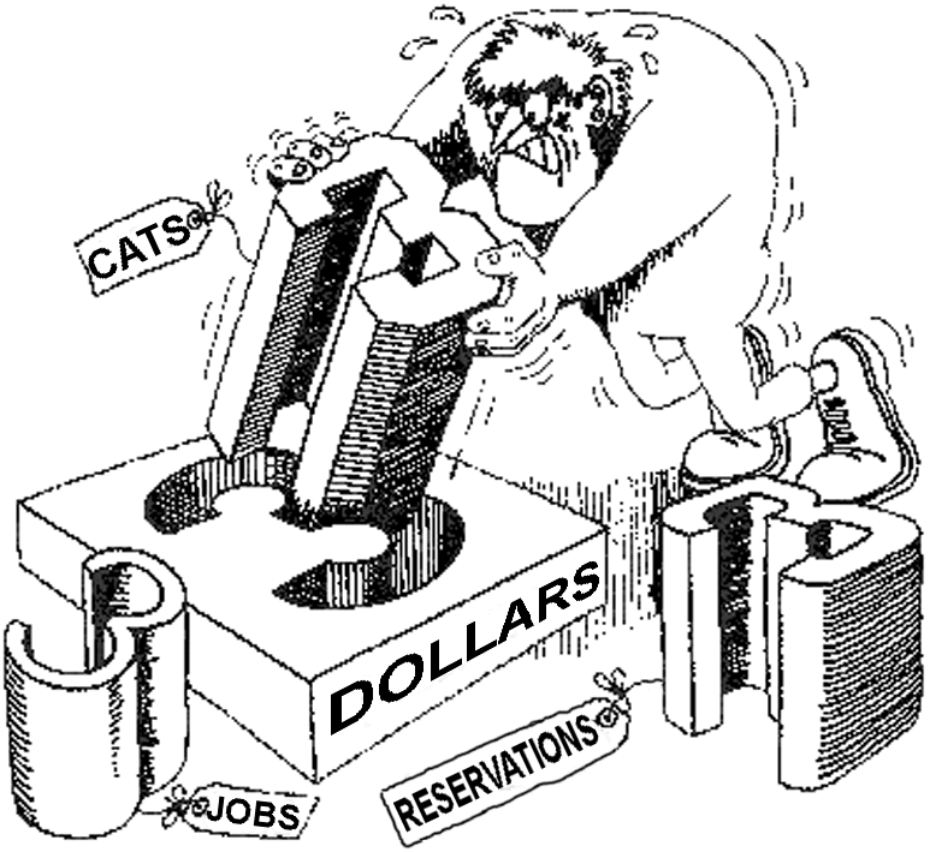


Рис. 2.9. Строгая типизация предотвращает смешивание абстракций

Строгая типизация в объектно-ориентированном проектировании предотвращает смешивание абстракций.

Языки программирования могут иметь механизмы типизации, а могут и не иметь.

2.3.6. Параллелизм

Для задач, в которых одновременно обрабатываются несколько событий или потребность в вычислительной мощности превышает ресурсы одного процессора, используют несколько компьютеров или задействуют многозадачность на многопроцессорном компьютере.

Параллельность достигается лишь на многопроцессорных системах. Системы с одним процессором только имитируют параллельность за счет алгоритмов разделения времени.

Объектно-ориентированные системы хорошо подходят для параллельной обработки. Параллелизм позволяет различным объектам действовать одновременно.

2.3.7. Сохраняемость

Любой программный объект существует в памяти и во времени. Спектр сохраняемости объектов охватывает следующий диапазон:

1) промежуточные результаты вычисления выражений;
2) локальные переменные в вызове процедур;
3) собственные переменные, глобальные переменные, динамические, создаваемые данные;
4) данные, сохраняющиеся между сеансами выполнения программ;
5) данные, которые нужно сохранять при переходе на новую версию программы;
6) данные, которые нужно сохранять даже после устаревания версии программы.

Традиционно первыми тремя уровнями занимаются языки программирования, а последними тремя - базы данных. Однако сейчас программисты используют специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных внедряют в свою технологию коротко живущие объекты.

Сохраняемость – это способность объектов существовать во времени, переживая породивший их процесс, и пространстве, перемещаясь из своего первоначального адресного пространства.

2.4. Объекты

И инженер, и художник должны хорошо чувствовать материал, с которым они работают. В объектно-ориентированной методологии анализа и создания сложных программных систем основными строительными блоками являются классы и объекты.

Объектом называют нечто обладающее состоянием, поведением и идентичностью. Структура и поведение схожих объектов определяет общий для них класс. Термины «экземпляр класса» и «объект» взаимозаменяемы. Понятия класса и объекта связаны, однако существует важное различие между ними. Объект обозначает конкретную сущность, определенную во времени и в пространстве. Класс определяет абстракцию существенного в объекте (его данные и поведение). Например, дом №2 по улице Лесной в нашем городе - объект класса House (Дом). Класс House определяет, что у дома должны быть высота, ширина, количество комнат. Объект этого класса - дома №2 по улице Лесной - может иметь высоту 20 футов, ширину 60 футов, 10 комнат. Класс - более общий термин, являющийся, по существу, шаблоном для объектов. Представьте себе, что класс - это проект дома, а объекты - это 5 построенных по проекту домов. ***Класс*** - это некое множество объектов, имеющих общую структуру и общее поведение.

Объектами могут быть осязаемые и видимые предметы (например, дом, цветок и т.п.).

Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение.

Объекты могут получаться из отношений между другими объектами. Два тела, например, сфера и куб, имеют, как правило, нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.

2.4.1. Состояние

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

К числу ***свойств*** объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Однако в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Все свойства имеют значения, которые могут изменяться. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом три, означающим номер этажа, на котором лифт в данный момент находится.

2.4.2. Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений (операций).

Иными словами, поведение объекта - это его наблюдаемая и проверяемая извне деятельность.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append (добавить) и pop (изъять элемент) длятого, чтобы управлять объектом-очередью. Существует также операция length, которая позволяет определить размер очереди, но не может изменить это значение. В основном понятие ***сообщение*** совпадает с понятием ***операции*** над объектами, хотя механизм передачи различен. В объектно-ориентированных языках операции, выполняемые над данным объектом, называются ***методами*** и входят в определение класса объекта.

Передача сообщений (операции) - это одна часть уравнения, задающего поведение. Из нашего определения следует, что состояние объекта также влияет на его поведение. Рассмотрим торговый автомат. Мы можем сделать выбор, но поведение автомата будет зависеть от его состояния. Если мы не опустили в него достаточную сумму, скорее всего ничего не про-

изойдет. Если же денег достаточно, автомат выдаст нам желаемое (и тем самым изменит свое состояние).

Итак, поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Концепция побочного действия позволяет уточнить наше определение состояния: ***состояние объекта*** представляет суммарный результат его поведения.

Наиболее интересны те объекты, состояние которых не статично, а изменяется и запрашивается операциями.

Для примера опишем на языке C++ класс Queue (очередь):

```
class Queue {
public:
   Queue();
   Queue(const Queue&);
   virtual ~Queue();
   virtual Queue& operator=(const Queue&);
   virtual int operator==(const Queue&) const;
   int operator!=(const Queue&) const;
   virtual void clear();
   virtual void append(const void*);
   virtual void pop();
   virtual void remove(int at);
   virtual int length() const;
   virtual int isEmpty() const;
   virtual const void* front() const;
   virtual int location(const void*);
protected:
...
};
```

В определении класса используется обычная для C идиома ссылки на данные неопределенного типа с помощью void*, благодаря чему в очередь можно вставлять объекты разных классов. Эта техника небезопасна, клиент должен ясно понимать, с объектом какого класса он имеет дело. Кроме того, при использовании void* очередь не «владеет» объектами, которые в нее помещены. Деструктор ~Queue() уничтожает очередь, но не ее участников (параметризованные типы позволяют справляться с такими проблемами).

Так как определение Queue задает класс, а не объект, мы должны объявить экземпляры класса, с которыми могут работать клиенты:

```
Queue a, b, c, d;
```

Можно выполнить следующие операции над объектами:

```
a.append(&deb);
a.append(&karen);
a.append (&denise);
b = a;
a.pop();
```

Теперь очередь a содержит двух сотрудников (первой стоит karen), а очередь b - трех (первой стоит deb). Таким образом, очереди имеют определенное состояние, которое влияет на их будущее поведение. Например, одну очередь можно безопасно продвинуть (pop) еще два раза, а вторую - три.

С точки зрения класса объекта ***операция*** - это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами операции пяти видов (в разделе третей главы, который посвящен операциям, вы найдете несколько иную классификацию: *операции реализации, операции управления, операции доступа и вспомогательные операции*). Ниже приведены три наиболее распространенные операции:

- ***модификатор*** - операция, которая изменяет состояние объекта;
- ***селектор*** - операция, считывающая состояние объекта, но не меняющая состояния;
- ***итератор*** - операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.

Кроме них существуют две универсальные операции. Они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса. Это:

- ***конструктор*** - операция создания объекта и/или его инициализации;
- ***деструктор*** - операция, освобождающая состояние объекта и/или разрушающая сам объект.

Операции могут быть не только методами класса, но и независимыми от объектов свободными подпрограммами. Свободные подпрограммы группируются в соответствии с классами, для которых они создаются. Это дает основание называть такие пакеты процедур утилитами класса. Например, для определенного выше класса Queue можно написать следующую свободную процедуру:

```
void copyUntilFound(Queue& from, Queue& to, void* item)
{
   while ((!from.isEmpty()) && (from.front() != item)) {
         to.append(from.front());
         from.pop();
         }
}
```

Смысл в том, что содержимое одной очереди переходит в другую до тех пор, пока в голове первой очереди не окажется заданный объект. Это операция высокого уровня, она строится на операциях-примитивах класса Queue.

Таким образом, можно утверждать, что все методы - операции, но не все операции - методы: некоторые из них представляют собой свободные подпрограммы.

Совокупность всех методов и свободных подпрограмм, относящихся к конкретному объекту, образует ***протокол*** этого объекта. Протокол, таким образом, определяет поведение объекта, охватывающее все его статические и динамические аспекты. В самых нетривиальных абстракциях полезно подразделять протокол на частные аспекты поведения, которые мы будет называть ***ролями***.

Объединяя определения состояния и поведения объекта, вводят понятие ответственности. ***Ответственность объекта*** имеет две стороны - знания, которые объект поддерживает, и действия, которые объект может исполнить. Они выражают смысл его предназначения и место в системе. Ответственность понимается как совокупность всех услуг и всех контрактных обязательств объекта. Таким образом, можно сказать, что состояние и поведение объекта определяют исполняемые им роли, а те, в свою очередь, необходимы для выполнения ответственности данной абстракции.

Большинство объектов исполняют в своей жизни разные роли, например:

1) банковский счет может быть в хорошем или плохом состоянии (две роли), и от этой роли зависит, что произойдет при попытке снятия с него денег;

2) для фондового брокера пакет акций - это товар, который можно покупать или продавать, а для юриста это знак обладания определенными правами;

3) в течение дня одна и та же персона может играть роль матери, врача, садовника и кинокритика.

Роли банковского счета являются динамическими и взаимоисключающими. Роли пакета акций слегка перекрываются, но каждая из них зависит от того, что клиент с ними делает. В случае персоны роли динамически изменяются каждую минуту.

Мы часто начинаем анализ задачи с перечисления разных ролей, которые может играть объект. Во время проектирования мы выделяем эти роли, вводя конкретные операции, выполняющие ответственности каждой роли.

2.4.3. Идентичность

Идентичность - это такое свойство объекта, которое отличает его от всех других объектов.

Не следует путать идентичность объекта с его ключевыми атрибутами, именем или номером.

2.4.4. Время жизни объекта

Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием - возвращение отведенного участка памяти системе.

Объекты создаются явно или неявно. Есть два способа создать их явно. Во-первых, это можно сделать при объявлении, тогда объект размещается в стеке. Во-вторых, можно разместить объект, выделив ему память из «кучи». В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной памяти под объект.

Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта. Более того, создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики копирующего конструктора и оператора присваивания в C++ разрешает явное управление тем, когда части объекта создаются и уничтожаются. К тому же в C++ можно переопределять и оператор new, тем самым изменяя политику управления памятью в «куче» для отдельных классов.

При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами (Деструкторы не освобождают автоматически память, размещенную оператором new, программисты должны явно освободить ее).

2.5. Классы

Понятия класса и объекта тесно связаны: невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие между этими понятиями. Объект обозначает конкретную сущность, определенную во времени и в пространстве. класс определяет данные и поведение, которыми должен обладать объект. Например, для объекта дом №2 по улице Лесной можно создать класс House (Дом), который будет определять, что у дома должны быть высота, ширина, длинна и количество квартир. Тогда у объекта класса House - дома №2 по улице Лесной - могут быть высота 15 метров, ширина 10 метров, длинна 50 метров и 60 квартир. Класс - более общий термин, являющийся, по существу, шаблоном для объектов. Класс можно сравнить с проектом дома, а объекты - с построенными по проекту домами. Таким образом, ***класс*** - это некое множество объектов, имеющих общую структуру и общее поведение. Любой конкретный объект является экземпляром класса. Объект не является классом, хотя класс может быть объектом.

Класс изображают в виде прямоугольника, разделенного на три части (рис. 2.10). В верхней части прямоугольника содержится имя и (необязательно) стереотип класса. Средний раздел включает в себя атрибуты класса. В нижней секции описываются операции или поведение класса.

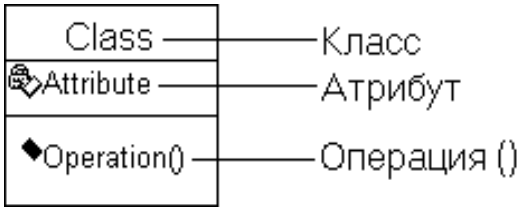


Рис. 2.10. Графическая нотация для изображения класса

<<Entity>> Employee
- Employee_ID:integer=0 # $ Salary:float + Address:string + State:string
Hire(new_dept:String, new_salary:float=50):integer Fire() + Transfer() + GetSalary()

Рис. 2.11. Пример класса Employee (Сотрудник)

Атрибут – это некоторая информация, характеризующая класс. Например, класс Employee (Сотрудник) (рис. 2.11) имеет четыре атрибута: Employee_ID (Идентификационный номер), Salary (Оклад), Address (Адрес), State (Статус). ***Операции*** класса отражают его поведение (действия, выполняемые классом). Для класса Account определены четыре операции: Hire (Нанять), Fire (Уволить), Transfer (Перевести), GetSalary (Установить оклад).

2.5.1. Типы и стереотипы классов

Кроме регулярных классов доступны классы следующих типов: параметризованные, классы-наполнители, утилиты классов, утилиты параметризованных классов, утилиты классов-наполнителей, метаклассы.

Параметризованный класс (parameterized class) применяется для создания семейства других классов, его еще называют шаблоном (контейнером).

Параметризованный класс изображается с помощью нотации показанной на рис. 2.12, а. В прямоугольнике, выделенном пунктирными линиями, указываются аргументы параметризованного класса. Изменяя значения аргументов, мы получаем стандартные классы. Например, параметризованный класс List (Список) позволяет создать такие классы, как EmployeeList (Список сотрудников), OrderList (Список заказов) и AccountList (Список счетов). Для этого в приведенной выше нотации нужно заменить параметр «Элемент» на соответствующий специфический элемент Employee (Сотрудник), Order (Заказ) или Account (Счет).

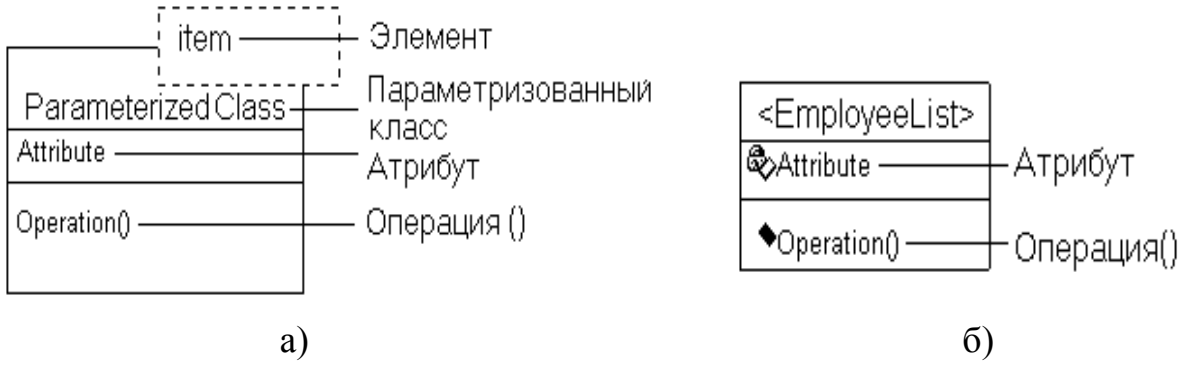


Рис. 2.12. Графическая нотация для изображения:
а) параметризованного класса; б) класса-наполнителя

Аргументом параметризованного класса может быть другой класс, тип данных или выражение-константа. Можно задавать неограниченное количество аргументов.

Класс-наполнитель (instantiated class) является параметризованным классом, аргументы которого имеют фактические значения. Для рассмотренного выше примера это может быть класс EmployeeList (Спосок сотрудников). В соответствии с нотацией UML, название аргумента класса-наполнителя заключается в угловые скобки (< >) (рис. 2.12, б).

Если в системе есть совокупность функций, которые используются всей системой и не слишком хорошо подходят для какого-либо конкретного класса, эти функции можно объединить в ***утилиту класса*** (class utility), которая будет использоваться всеми классами системы. Утилиты классов часто применяют для расширения функциональных возможностей языка программирования или для хранения общих элементов функциональности многократного использования, необходимых в нескольких системах. На диаграмме утилита класса выглядит как класс с тенью (рис. 2.13).

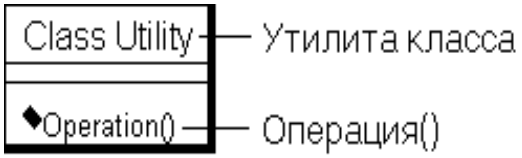
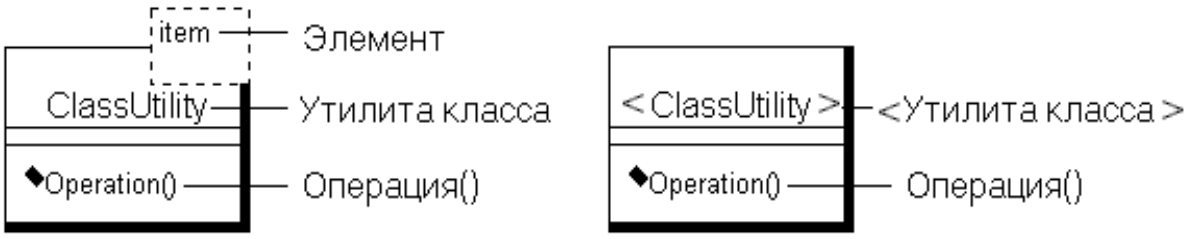


Рис. 2.13. Графическая нотация для изображения утилиты класса

Утилитой параметризованного класса (parameterized class utility) является параметризованный класс, содержащий только набор операций. Это шаблон для создания утилит класса. Утилита параметризованного класса изображается с помощью нотации показанной на рис. 2.14, а.



а) б)

Рис. 2.14. Графическая нотация для изображения:
а) утилиты параметризованного класса; б) утилиты класса-наполнителя

Утилитой класса-наполнителя (instantiated class utility) называется утилита параметризованного класса, параметры которой имеют фактические значения. Утилита класса-наполнителя изображается с помощью нотации показанной на рис. 2.14, б.

Метакласс (metaclass) - это класс, экземпляры которого являются классами, а не объектами. К числу метаклассов относятся параметризован-

ные классы и утилиты параметризованных классов. Метаклассы изображают с помощью нотации показанной на рис.2.15.

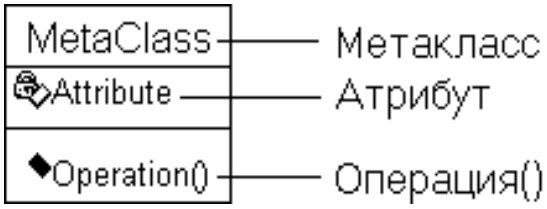


Рис. 2.15. Графическая нотация для изображения метакласса

Абстрактным называется класс, который не наполняется конкретным содержанием (не инстанцируется). Иными словами, если класс А абстрактный, в памяти никогда не будет объектов типа А.

Обычно абстрактные классы применяют при работе с наследованием. В них содержатся данные и поведение, общие для нескольких других классов. Например, у нас может быть класс Животное с атрибутами Рост, Цвет и Разновидность. От этого класса наследуют три других класса: Кошка, Собака и Птица. Каждый из них наследует свойства Рост, Цвет и Разновидность от класса Животное, а также имеет свои собственные уникальные атрибуты и операции. Объекты класса Животное не создаются во время работы приложения - все объекты являются только кошками, собаками и птицами. Класс Животное является абстрактным, он описывает, что общего есть у кошек, собак и птиц. В нотации UML название абстрактного класса пишут курсивом (рис.2.16).

Рис. 2.16. Графическая нотация для изображения абстрактного класса

Стереотип - это механизм, позволяющий классифицировать классы. Допустим, Вы хотите найти в модели все формы. Для этого можно создать стереотип Form (Форма) и назначить его всем окнам приложения. В дальнейшем при поиске форм нужно искать только классы с этим стереотипом. На языке UML определены три основных стереотипа: Boundary (Граница), Entity (Объект) и Control (Управление).

Пограничными классами (boundary classes) называются такие классы, которые расположены на границе системы со всем остальным миром. Они включают в себя формы, отчеты, интерфейсы с аппаратурой (принтер,

сканер и т.п.) и интерфейсы с другими системами. Пограничные классы изображают с помощью нотации показанной на рис. 2.17, а.

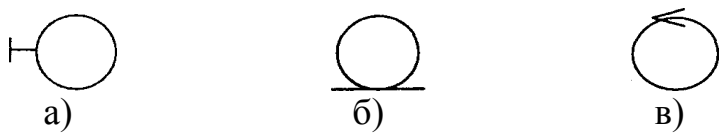


Рис 2.17. Графическая нотация для изображения:
а) пограничного класса; б) класса-сущности; в) управляющего класса

Для выявления пограничных классов необходимо исследовать диаграммы Вариантов Использования. Для каждого взаимодействия между действующим лицом и вариантом использования должен существовать хотя бы один пограничный класс (рис. 2.18, а). Именно он позволяет действующему лицу взаимодействовать с системой. Необязательно создавать уникальные пограничные классы для каждой пары «действующее лицо - вариант использования». Например, если два действующих лица инициируют один и тот же вариант использования, они могут применять для взаимодействия с системой общий пограничный класс (рис. 2.18, б).

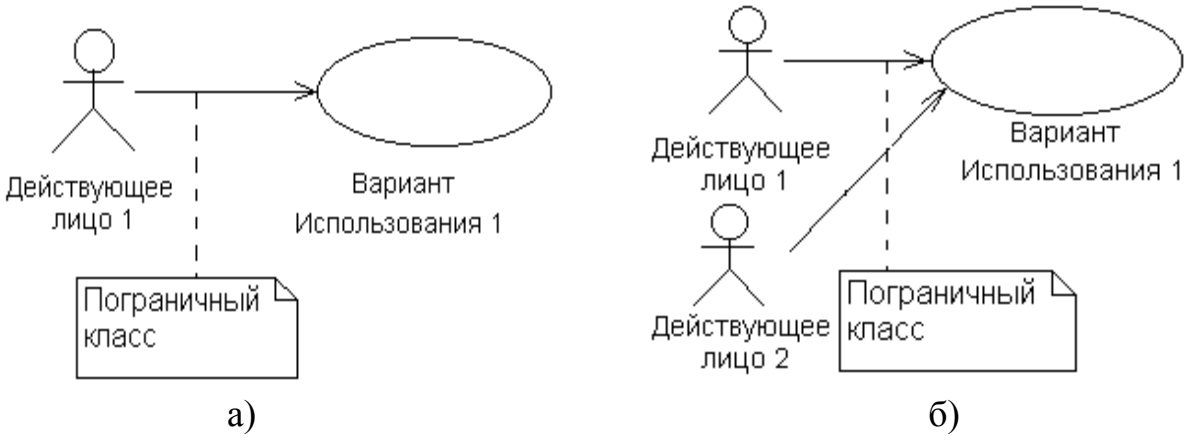


Рис. 2.18. Графическая нотация для изображения:
а) пограничного класса одного варианта использования; б) общего пограничного класса

Классы-сущности (entity classes) содержат информацию, хранимую постоянно. Классы-сущности можно обнаружить в потоке событий и на диаграммах Взаимодействия. Они имеют наибольшее значение для пользователя, и потому в их названиях часто применяют термины из предметной области. В системе работы с данными о сотрудниках примером такого класса является класс Employee (Сотрудник). Классы-сущности изображают с помощью нотации показанной на рис. 2.17, б.

Объектно-ориентированный подход может применяться для создании таблиц баз данных. Вместо того чтобы с самого начала задавать струк-

туру базы данных, ее разрабатывают на основе информации, получаемой из объектной модели. Из требований к системе определяют потоки событий. Из потоков событий определяют объекты, классы и атрибуты классов. Каждый атрибут класса-сущности становится полем в базе данных. Применяя такой подход, легко отслеживать соответствие между полями базы данных и требованиями к системе, что уменьшает вероятность сбора ненужной информации.

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. Сам управляющий класс не несет в себе никакой функциональности, другие классы посылают ему мало сообщений, но сам он посылает множество сообщений. Управляющий класс делегирует ответственность другим классам. По этой причине управляющий класс часто называют классом-менеджером. Управляющие классы изображают с помощью нотации показанной на рис. 2.17, в.

На рис. 2.19 приведен пример диаграммы Взаимодействия с управляющим классом ControlObject, который отвечает за координацию.

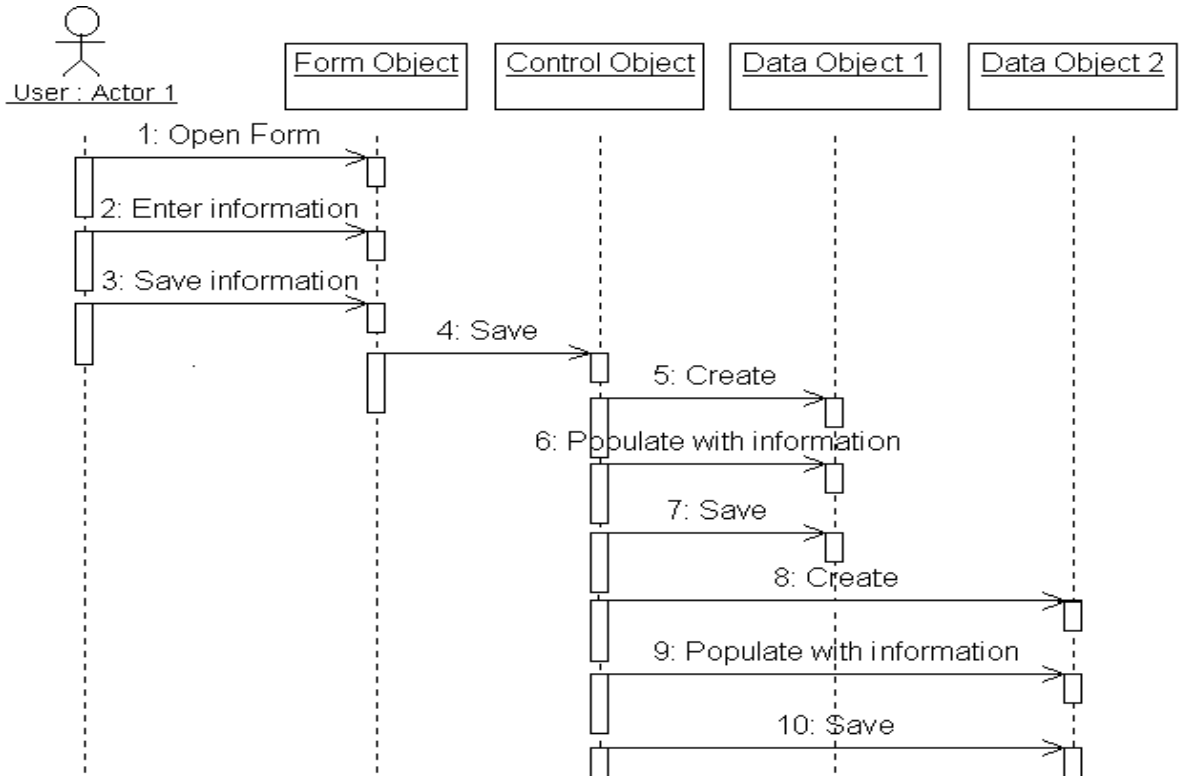


Рис. 2.19. Управляющий класс на диаграмме Последовательности

В системе могут применяться управляющие классы, общие для нескольких вариантов использования. Например, класс SecurityManager (Менеджер безопасности) может отвечать за контроль событий, связанных с

безопасностью, а класс TransactionManager (Менеджер транзакций) - заниматься координацией сообщений, относящихся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с такими элементами функционирования системы, как разделение ресурсов, распределенная обработка данных и обработка ошибок. С помощью управляющих классов можно изолировать функциональность системы. Инкапсуляция в один класс, например координации безопасности, минимизирует последствия вносимых изменений. Любые изменения логики системы отвечающей за безопасность затронут только менеджера безопасности.

2.5.2. Подходы к выявлению ключевых абстракций системы

Рассмотрим несколько подходов к анализу объектно-ориентированных систем с целью выявления ключевых абстракций.

1. Уэнди и Майкл Боггс в книге «Mastering UML with Rational Rose» предлагают ***подход к выявлению ключевых абстракций системы, основанный на использовании UML.***

Выявление классов можно начать с изучения имен существительных в описании потоков событий [2]. В общем случае имена существительные могут соответствовать следующим элементам: действующее лицо, класс и атрибут класса.

Классы также можно обнаружить, анализируя диаграммы Взаимодействия. На диаграммах нужно определить похожие объекты и создать для них общий класс. Допустим, диаграмма Последовательности описывает процесс выплаты зарплаты Джону и Фреду. Атрибуты обоих объектов Джон и Фред сходны: имя, адрес и номер телефона. Операции тоже одинаковы: нанять или уволить сотрудника. Таким образом, можно создать класс Employee (Сотрудник), который будет шаблоном для объектов Джон и Фред.

Не все классы можно обнаружить в потоках событий и на диаграммах Взаимодействия. При выявлении классов следует также рассмотреть три возможных стереотипа: сущность (entity), границу (boundary) и управление (control).

2. Разные ученые предлагают свои источники классов и объектов. Такие подходы считаются ***классическими подходами к выявлению ключевых абстракций системы.***

Шлеер и Меллор предлагают следующих кандидатов в классы и объекты:

1) осязаемые предметы (автомобили, телеметрические данные, датчики давления и т.п.);
2) роли (мать, учитель, политик и т.п.);
3) события (посадка, прерывание, запрос и т.п.);

4) взаимодействие (перечисление, пересечение, заем и т.п.).

Роусс, исходя из моделирования баз данных, предлагает следующих кандидатов в классы и объекты:

1) люди (человеческие существа, выполняющие некоторые функции);
2) места (области связанные с людьми или предметами);
3) предметы (осязаемый материальный объект или группы объектов);
4) организации (формально организованная совокупность людей, ресурсов, оборудования, которая имеет определенную цель и существование которой от индивидуумов не зависит);
5) концепции (принципы и идеи, не осязаемые сами по себе, но предназначенные для организации деятельности, общения или для наблюдения за ними);
6) события (нечто случающееся с чем-то в заданное время или последовательно).

Коад, Йордан предлагают следующих кандидатов в классы и объекты:

1) структуры (отношение «целое-часть» и «общее-частное»);
2) другие системы (внешние системы, с которыми взаимодействует приложение);
3) устройства, с которыми взаимодействует приложение;
4) события (происшествия, которые должны быть запомнены);
5) разыгрываемые роли (роли, которые исполняют пользователи, работающие с приложением);
6) места (здания, офисы и другие места, существенные для работы приложения);
7) организационная единица (группы, к которым принадлежат пользователи).

На более высоком уровне абстракции Коад вводит понятие предметной области, которая в сущности является логически связанной группой классов, относящейся к высокоуровневым функциям системы.

3. В то время как классические подходы концентрируют внимание на осязаемых элементах предметной области, ***анализ поведения*** рассматривает в качестве первоисточника объектов и классов динамическое поведение системы. Классы формируют, основываясь на группах объектов, демонстрирующих сходное поведение.

Вирфс-Брок предлагает следующую схему формирования классов. Объекты, которые имеют сходные ответственности, объединяются в общий класс. В иерархии классов каждый подкласс, выполняя обязательства суперкласса, может привносить свои дополнительные услуги.

Рубин и Гольдберг предлагают идентифицировать классы и объекты, анализируя функционирование системы: «Наш подход основан на изуче-

нии поведения системы. Мы сопоставляем формы поведения с частями системы и пытаемся понять, какая часть инициирует поведение, и какие части в нем участвуют... Инициаторы и участники, играющие существенные роли, опознаются как объекты и делаются ответственными за эти роли» [3].

Идеи Рубина тесно связаны с подходом, предложенным в 1979 году Альбрехтом. Данный проход ориентирован на функции. По Альбрехту, функция определяется как «отдельное бизнес-действие конечного пользователя», то есть: ввод/вывод, запрос, файл или интерфейс [3]. Эта концепция происходит из области информационных систем, однако, может быть применена к любой автоматизированной системе. По существу, функция - это любое достоверно видимое извне и имеющее отношение к делу поведение системы.

4. Еще один подход к выявлению ключевых абстракций системы основан на ***анализе предметной области***.

Иногда в поисках полезных и доказавших свою работоспособность идей имеет смысл обратиться сразу к нескольким приложениям в рамках рассматриваемой предметной области. Если Вы испытываете затруднения в процессе разработки ПО, исследуйте уже имеющиеся подобные системы. Это поможет понять, какие ключевые абстракции и механизмы, использованные в них, будут полезны, а какие нет. В качестве примера рассмотрим систему бухгалтерского учета. Допустим, она должна представлять различные виды отчетов. Если считать работу с отчетами предметной областью, ее анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты будут представлять собой множество ключевых абстракций и механизмов, отобранных с учетом одной из целей исходной задачи - создания системы отчетов.

Идею анализа предметной области впервые предложил Нейборс. Такой анализ определяют еще, как «попытку выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными» [3]. Мур и Байлин определяют следующие этапы анализа предметной области:

1) «построение скелетной модели предметной области при консультации с экспертами в этой области;
2) изучение существующих в данной области систем и представление результатов в стандартном виде;
3) определение сходства и различий между системами при участии экспертов;
4) уточнение общей модели для приспособления к нуждам конкретной системы» [3].

В роли эксперта часто выступают обычные пользователи системы, например, инженер или диспетчер. Эксперт не обязательно должен быть

программистом, он должен быть близко знаком с исследуемой областью и «разговаривать» на ее языке. Сотрудничество пользователей и разработчиков системы имеет важное значение.

Анализ предметной области лучше вести попеременно с проектированием - немного поанализировать, затем немного попроектировать и т.д. Данный подход достаточно эффективен, исключая лишь узко специализированные области, но такие уникальные программные системы встречаются редко.

5. В отдельности классический подход, поведенческий подход и изучение предметной области, рассмотренные выше, сильно зависят от индивидуальных способностей и опыта аналитика. Для большинства реальных проектов одновременное применение всех трех подходов неприемлемо, так как процесс анализа становится недетерминированным и непредсказуемым. ***Анализ вариантов*** - это подход, который можно успешно сочетать с перечисленными, делая их применение более упорядоченным. Впервые его формализовал Джекобсон, определивший вариант применения, как «частный пример или образец использования, сценарий, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий» [3].

Опишем этапы проведения анализа вариантов. Этот вид анализа можно начинать вместе с анализом требований, когда пользователи, эксперты и разработчики перечисляют сценарии, наиболее существенные для работы системы (пока не углубляясь в детали). Затем сценарии тщательно прорабатывают, раскладывая их по кадрам, как это делают телевизионщики и кинематографисты [3]. При этом устанавливается, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Тем самым четко распределяются области влияния абстракций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение. В результате появляются новые или уточняются существующие абстракции.

6. Следующий подход к выявлению ключевых абстракций системы использует ***CRC-карточки***. CRC обозначает Class-Responsibilities-Collaborators (Класс/Ответственности/Участники). Это простой и эффективный способ анализа сценариев. Карты CRC впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию, но такие карточки оказались хорошим инструментом для «мозгового штурма» и общения разработчиков между собой.

CRC-карты – это обычные библиографические карточки размером 3 на 5 или 5 на 7 дюймов. Хорошо, если карточки будут линованными и разноцветными. На карточках записывают карандашом сверху - название класса, снизу в левой половине - за что он отвечает, а в правой половине - с кем он взаимодействует. Проходя по сценарию, заводят по карточке на каждый обнаруженный класс. Полученные абстракции анализируют, при

этом можно либо выделить излишек ответственности в новый класс, либо перенести ответственности с одного большого класса на несколько более детальных классов, либо передать часть обязанностей другому классу.

Карточки позволяют представить разные формы взаимодействия объектов. С точки зрения динамики сценария их расположение может показать поток сообщений между объектами, с точки зрения статики они представляют иерархии классов.

7. ***Метод использования неформального описания задачи*** с целью выявления ключевых абстракций системы впервые был предложен Аббо-том. Согласно ему, нужно описать задачу или ее часть на простом русском языке, а потом подчеркнуть имена существительные и глаголы [3]. Имена существительные - кандидаты на роль классов, а глаголы могут стать именами операций.

Данный метод прост и заставляет разработчиков заниматься словарем предметной области. Однако он весьма приблизителен и не подходит для сложных проблем. Человеческий язык – довольно неточное средство выражения, поэтому список объектов и операций зависит от умения разработчика записывать свои мысли. Кроме этого для многих существительных можно найти соответствующую глагольную форму и наоборот.

8. Еще один способ использует в качестве источника ключевых абстракций системы продукты ***структурного анализа***. После проведения структурного анализа уже существует модель системы, описанная диаграммами потоков данных и другими продуктами структурного анализа. На основе этих моделей можно определить классы и объекты тремя различными способами.

Мак Менамин и Палмер предлагают сначала сформировать словарь данных, а затем приступить к анализу контекстных диаграмм модели. Они утверждают следующее: «Рассматривая список основных структур данных, следует подумать, что они описывают. Например, если они имена прилагательные, то какие имена существительные они описывают? Ответы на такие вопросы могут пополнить список объектов» [3]. Источниками объектов в этом случае являются: предметная область, входные и выходные данные, услуги и других ресурсы.

Следующий способ основан на анализе диаграмм потоков данных. В этом случае кандидатами в объекты могут быть:

- внешние сущности;
- хранилища данных;
- хранилища управляющих сущностей;
- управляющие преобразования.

Кандидатами в классы являются потоки данных и потоки управления. Преобразование данных можно рассматривать как операции над объектами или как поведение некоторого объекта.

Зайдевиц и Старк предлагают еще один метод, который они называют анализом абстракций. В структурном анализе входные и выходные данные изучаются до тех пор, пока не достигнут наивысшего уровня абстракции. Процесс преобразования входных данных в выходные рассматривается как основное преобразование. В абстрактном анализе разработчик изучает это преобразование для того, чтобы определить, какие процессы и состояния являются самыми важными. Так определяются основные сущности. Затем с помощью диаграмм потоков данных изучают всю инфраструктуру, прослеживая входящие и исходящие из центра потоки данных, и группируют встречающиеся на пути процессы и состояния.

Следует отметить, что принципы структурного проектирования, который основан на структурном анализе, полностью ортогональны принципам объектно-ориентированного проектирования. Диаграммы потоков данных представляют собой скорее описание проекта, чем модель существа системы. Кроме этого трудно построить объектно-ориентированную систему, если модель ориентирована на алгоритмическую декомпозицию. Поэтому лучше использовать как подготовительный этап для объектно-ориентированного проектирования объектно-ориентированный анализ.

2.5.3. Атрибуты

Атрибут - это фрагмент информации связанный с классом. Рассмотрим основные типы, характеристики и способы выявления атрибутов.

С атрибутами можно связать три основных фрагмента информации: имя атрибута, тип его данных и первоначальное значение.

Тип данных атрибута специфичен для используемого языка. Это может быть, например, тип string, integer, long или boolean.

Рассмотрим пример определения первоначального значения (значения по умолчанию). Допустим, атрибут TaxRate (Ставка налога с покупки) класса Order (Заказ) для какого-либо города равняется 7.5%. Следовательно, для него можно определить значение по умолчанию, равное 0.075. Значение по умолчанию атрибута задавать не обязательно. В описании атрибута оно указывается после знака «=» (рис. 2.11).

При добавлении атрибута к классу каждый экземпляр класса получит свою собственную копию этого атрибута. Рассмотрим, например, класс Employee. В процессе работы приложения можно создать экземпляры двух сотрудников Джона и Билла. Каждый из этих объектов получит свою собственную копию атрибута Salary (Оклад). ***Статичный атрибут*** (static) - это такой атрибут, который используется всеми экземплярами класса. Если бы атрибут Salary был статичным, он был бы общим для Джона и Билла. На языке UML статичный атрибут помечают символом «$». В нашем примере Salary станет «$Salary» (рис. 2.11).

Производным (derived) называется атрибут, полученный из одного или нескольких других атрибутов. Например, класс Rectangle (Прямоугольник) может иметь атрибуты Width (Ширина) и Height (Высота). У него также может быть атрибут Area (Площадь), вычисляемый как произведение ширины и высоты. Так как Area получается из этих двух атрибутов, он считается производным атрибутом. В нотации UML производные атрибуты помечают символом «/». В описанном примере атрибут Area следует написать как «/Area» (рис. 2.20).

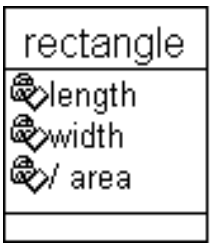


Рис. 2.20. Графическая нотация для изображения производного атрибута

Рассмотрим способы выявления атрибутов.

Один из источников атрибутов - имена существительные в потоке событий. Некоторые из них являются классами или объектами, другие - действующими лицами, третьи - атрибутами. Например, в потоке событий может быть написано: «Пользователь вводит имя сотрудника, его адрес, номер социальной страховки и номер телефона». Это означает, что у класса Сотрудник имеются атрибуты Имя, Адрес, Номер страховки и Номер телефона.

Атрибуты можно также выявить, изучая документацию, описывающую требования к системе. Нужно изучить такие требования, которые определяют собираемые системой данные. Любой элемент такой информации может быть атрибутом класса.

Еще одним источником атрибутов может стать структура базы данных. Если она уже определена, поля в ее таблицах дадут хорошее представление об атрибутах. Часто имеется однозначное соответствие между таблицами базы данных и классами-сущностями. Если вернуться к предыдущему примеру, то таблица Сотрудник будет иметь поля: Имя, Адрес, Номер телефона и Номер страховки. Тогда соответствующий класс Сотрудник будет иметь атрибуты: Имя, Адрес, Номер телефона и Номер страховки. Однако не всегда существует такое однозначное соответствие. Подходы к проектированию баз данных и классов могут различаться. В частности, реляционные базы данных не поддерживают наследование.

Определяя атрибуты, нужно следить за тем, чтобы каждый из них можно было соотнести с требованиями к системе. Если это не удается сделать, нельзя быть уверенным в том, что данное требование нужно за-

казчику. Такой подход помогает решить классическую проблему приложения, собирающего огромный объем никому не нужной информации.

После определения атрибутов их нужно соотнести с классами. Например, класс Сотрудник может содержать имя и адрес, но не должен включать в себя сведения о выпускаемой компанией продукции. Для последних подошел бы класс Продукция.

Обратите внимание на классы, у которых слишком много атрибутов. Возможно, такой класс следует разделить на два меньших. Так, класс с десятью или пятнадцатью атрибутами может быть вполне приемлемым. Только нужно убедиться, что все его атрибуты нужны и действительно должны принадлежать этому классу. Будьте осторожны с классами, у которых слишком мало атрибутов. Вполне возможно, что все нормально - например, управляющий класс имеет мало атрибутов. Однако это может быть и признаком необходимости в объединении нескольких классов.

Иногда могут возникнуть сомнения, соответствует ли обнаруженная Вами информация атрибуту или классу. Рассмотрим, например, такой атрибут, как название компании. Является ли он атрибутом класса Person (Человек), или лучше создать отдельный класс Company (Компания)? Ответ зависит от того, какое приложение Вы разрабатываете. Если Вы собираете сведения о компании и имеется связанное с ней поведение, она может быть классом. Допустим, что Вы проектируете систему работы с заказчиками. В таком случае может потребоваться информация о компаниях, которым поставляются товары или услуги. Тогда нужно знать, сколько сотрудников работает в компании, ее имя и адрес, контактный телефон и т.д. С другой стороны, специфическая информация о компании может и не требоваться. Допустим, приложение должно генерировать письма людям, работающим в других организациях. При этом достаточно знать названия их фирм. В таком случае имя компании будет атрибутом класса Контакт. Кроме того, нужно рассмотреть, есть ли поведение у подозрительной информации. Если в вашем приложении компания имеет некоторое выраженное поведение, лучше моделировать ее как класс. Если поведения нет, то это скорее всего атрибут.

2.5.4. Операции

Операцией называется связанное с классом поведение. Операции определяют ответственности классов.

Описание операции состоит из трех частей: имени, параметров и типа возвращаемого значения. Аргументы или параметры операции - это получаемые ею входные данные. Тип возвращаемого значения относится к результату действия операции.

В языке UML для операций принята следующая нотация:

имя операции (аргумент1:тип данных аргумента1, аргумент2:тип данных аргумента2, ...):тип возвращаемого значения операции

Для каждого аргумента должны быть заданы имя и тип данных. На диаграмме Классов аргументы и их типы указываются в скобках после имени операции (рис. 2.11). При желании для аргументов можно задавать их значения по умолчанию (рис. 2.11). В таком случае нотация UML будет иметь вид:

имя операции (аргумент1:тип данных аргумента1=значение по умолчанию аргумента1, …):тип возвращаемого значения операции

Выделяют четыре типа операций: операции реализации, операции управления, операции доступа и вспомогательные операции.

Операции реализации (implementor operations) реализуют некоторую бизнес-функциональность. Такие операции можно найти, исследуя диаграммы Взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциональности. Каждое сообщение такой диаграммы можно соотнести с операцией реализации.

Необходимо, чтобы каждую операцию реализации можно было проследить до соответствующего требования. Это достигается на различных этапах моделирования. Операция выводится из сообщения на диаграмме Взаимодействия, сообщения выделяются из подробного описания потока событий, который создается на основе варианта использования, а последний - на основе требований. Возможность проследить всю эту цепочку гарантирует, что каждое требование будет воплощено в коде, а каждый фрагмент кода реализует какое-то требование.

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

Атрибуты обычно бывают закрытыми или защищенными. Тем не менее другие классы иногда должны просматривать или изменять их значения. Для этого предназначены ***операции доступа*** (access operations). Допустим, имеется атрибут Salary (Оклад) класса Employee (Сотрудник). Было бы не желательно, чтобы другие классы могли изменять этот атрибут. Для выполнения этого условия добавим к классу Employee две операции доступа: GetSalary и SetSalary. К первой из них, являющейся общей, могут обращаться все классы. Она получает значение атрибута Salary и возвращает его вызвавшему ее классу. Операция SetSalary также является общей, она позволяет вызвавшему ее классу установить новое значение атрибута Salary. Эта операция может содержать любые правила и условия проверки, которые необходимо выполнить, прежде чем изменить атрибут. Операции доступа дают возможность безопасно инкапсулировать атрибуты внутри

класса, защищая их от других классов, но при этом позволяет осуществлять контролируемый доступ к ним.

Создание операций Get и Set (получения и изменения значения) для каждого атрибута класса является промышленным стандартом.

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.

Как и операции реализации, вспомогательные операции можно обнаружить на диаграммах Последовательности и Кооперативных диаграммах. Часто такие операции являются рефлексивными сообщениями.

При рассмотрении отдельных типов операций уже описывались способы их выявления, обобщим эти сведения.

Создав диаграммы Взаимодействия, Вы проделаете основную часть работы, требуемой для выявления операций.

Для идентификации операций выполните следующие действия.

1. Изучите диаграммы Взаимодействия. Большая часть сообщений на этих диаграммах является операциями реализации. Рефлексивные сообщения будут вспомогательными операциями.

2. Рассмотрите управляющие операции. Возможно, требуется добавить конструкторы и деструкторы.

3. Рассмотрите операции доступа. Для каждого атрибута класса, с которым будут работать другие классы, необходимо создать операции Get и Set.

При идентификации операций и анализе классов следует иметь в виду следующее.

1. Относитесь с подозрением к любому классу, имеющему только одну или две операции. Возможно, класс написан совершенно правильно, но его следует объединить с каким-нибудь другим классом.

2. С большим подозрением относитесь к классу без операций. Как правило, класс инкапсулирует не только данные, но и поведение. Класс без поведения лучше моделировать как один или несколько атрибутов.

3. С осторожностью относитесь к классу со слишком большим числом операций. Набор ответственностей класса должен быть управляем. Если класс очень большой, им будет трудно управлять. В такой ситуации лучше разделить класс на два меньших.

Как и в случае других элементов модели, для классификации операций создаются их ***стереотипы***. Существуют четыре наиболее распространенных стереотипа операций:

1) Implementor (операции реализации) - операции, реализующие некоторую бизнес-логику;

2) Manager (операции управления) - конструкторы, деструкторы и операции управления памятью;

3) Access (операции доступа) - операции, позволяющие другим классам просматривать или редактировать атрибуты данного класса. Как правило, такие операции называют GetИмя_атрибута или SetИмя_атрибута;

4) Helper (вспомогательные операции) - закрытые или защищенные операции, которые используются классом, но не видны другим классам.

Назначение операциям стереотипов облегчает понимание модели. Кроме того, стереотипы помогают убедиться в том, что ни одна операция не была пропущена. Стереотип указывается перед именем операции в двойных угловых скобках (<< >>), например, <<Entity>> (рис. 2.11).

2.5.5. Понятие видимости

Разделяют внутренне устройство классов - реализацию и внешнее устройство класса интерфейс. В ***интерфейсе*** главное - объявление операций, которые поддерживаются экземплярами классов. К интерфейсу также относятся объявление других классов, переменных, констант и ситуаций, уточняющих абстракцию. ***Реализация*** класса никому кроме самого класса недоступна, она состоит из реализации объявленных в интерфейсе операций.

С интерфейсом класса связана такая характеристика, как видимость. ***Видимость*** показывает, каким образом данные (атрибуты) и поведение (операции) инкапсулируются в класс. В таблице 2.1 приведены допустимые значения этого параметра (см. также рис. 2.11).

Таблица 2.1. Допустимые значения видимости атрибутов и операций

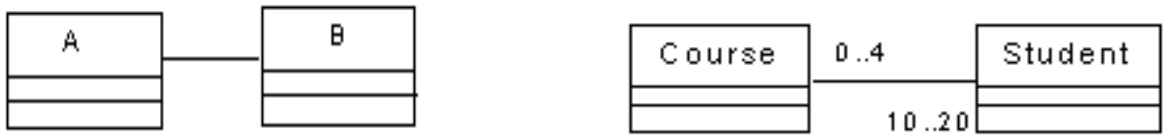
Тип видимости	Обозначение	Описание
public (общий, открытый)	«+»	Атрибут или операция доступные всем классам
private (закрытый)	«-»	Атрибут или операция не доступные другим классам
protected (защищенный)	«#»	Атрибут или операция доступные только самому классу и его потомкам
package or implementation (пакетный)		Атрибут или операция являются общими, но только в пределах своего пакета

2.5.6. Связи

2.5.6.1. Типы связей

Между классами могут быть установлены следующие типы связей: ассоциация, зависимость, агрегация и обобщение. Рассмотрим каждую из них подробнее.

1. ***Ассоциация*** - это семантическая связь между классами (то есть можно указать только роли, которые классы играют друг для друга). Ассоциация дает классу возможность узнавать об общих атрибутах и операциях другого класса. Ассоциацию часто используют на ранней стадии анализа, чтобы зафиксировать участников отношения. В дальнейшем ассоциации обычно уточняются, становясь более специализированной связью.



а) б)

Рис. 2.21. Графическая нотация для изображения:
а) ассоциации; б) множественности связи

После того, как классы связали ассоциацией, они могут передавать друг другу сообщения на диаграммах Взаимодействия. Ассоциации могут быть двунаправленными или однонаправленными. На языке UML двунаправленные ассоциации изображают в виде простой линии без стрелок (рис. 2.21, а) или со стрелками с обеих сторон. На однонаправленной ассоциации ставят только одну стрелку, показывающую направление связи.

Множественность (multiplicity) показывает, сколько экземпляров одного класса взаимодействуют с одним экземпляром другого класса в данный момент времени. Приведем пример. При разработке системы регистрации курсов в университете можно определить классы Course (Курс) и Student (Студент). Вопросы, на которые должен ответить параметр множественности: «Сколько курсов студент может посещать в данный момент?» и «Сколько студентов могут посещать один курс?». Индикаторы множественности устанавливаются на обоих концах линии связи. В нашем примере можно решить, что одному студенту разрешается посещать от 0 до 4 курсов, а один курс могут слушать от 10 до 20 студентов (рис. 2.21, б).

Для того чтобы уменьшить область действия ассоциации применяются ***квалификаторы*** (qualifiers). Допустим, что между классами Person и Company установлена связь ассоциации и что для данного значения атрибута Person ID (Идентификационный номер) существуют две взаимодей-

ствующие с классом компании. Это можно показать на диаграмме с помощью квалификаторов (рис. 2.22).

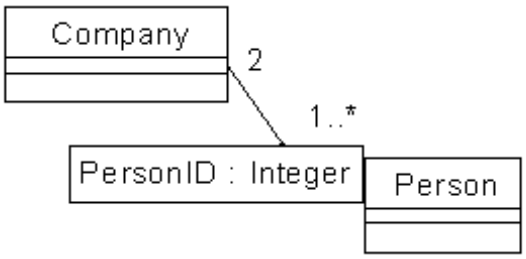


Рис. 2.22. Использование квалификаторов

2. ***Связь зависимости*** (использования) показывает, что один класс (клиент) ссылается на другой (сервер). Типичный случай проявления такого отношения - когда в реализации операции происходит объявление локального объекта используемого класса. Строгое отношение зависимости ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера. Зависимости всегда однонаправлены. Их изображают в виде стрелки, проведенной пунктирной линией (рис. 2.23).

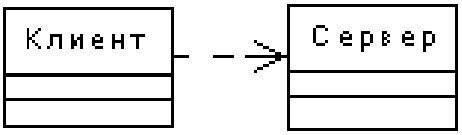


Рис. 2.23. Связь зависимости

3. ***Агрегация*** (aggregations) представляет собой более тесную форму ассоциации. Агрегация - это связь между целым и его частями. Например, такую связь можно установить между классом Автомобиль и классами для отдельных частей автомобиля, такими как Двигатель, Покрышки и т.п. Агрегацию изображают в виде линии с ромбиком у класса, являющегося целым (рис. 2.24, а).

а) б)

Рис. 2.24. Графическая нотация для изображения:
а) простой агрегации; б) рефлексивной агрегации

Как и ассоциации, агрегации могут быть рефлексивными (рис.2.24,б). Рефлексивные агрегации предполагают, что один экземпляр класса состоит из одного или нескольких экземпляров того же класса. Например, комбинируя ингредиенты при приготовлении пищи, можно получить ингредиенты для других блюд. Иначе говоря, каждый ингредиент состоит из других ингредиентов.

Один и тот же класс может быть агрегирован несколькими классами. Например, одна и та же книга в библиотеке может входить сразу в несколько каталогов.

Один класс может участвовать в нескольких отношениях агрегации с другими классами. Например, такую связь образует класс Car (Автомобиль) с классами для отдельных частей автомобиля, таких как Door (Дверь), Tire (Покрышка) и т.п. Автомобиль может иметь 4 двери, 4 покрышки и т.д. (рис. 2.25).

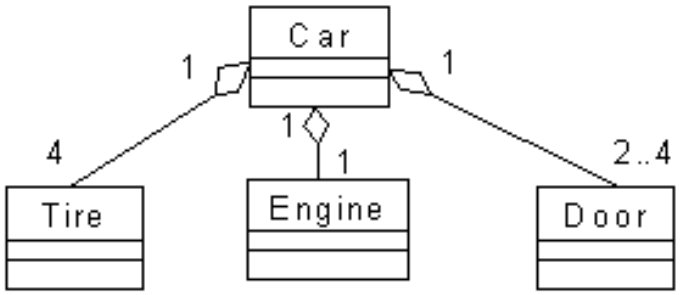


Рис. 2.25. Пример множественной агрегации

Пример агрегирования присутствует в простейшей файловой системе: каталоги могут содержать подкаталоги и файлы (рис. 2.26).

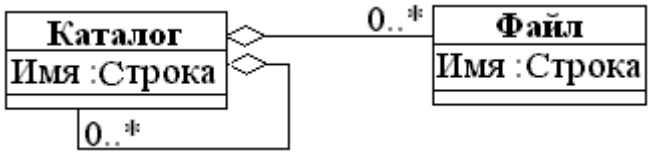


Рис. 2.26. Пример агрегирования

Если два класса связаны отношением агрегации, то в класс-целое войдут атрибуты для каждого класса-части. Можно устанавливать каким образом атрибуты будут включаться (метод включения): по значению или по ссылке. ***Агрегация по значению*** (By Value) предполагает, что целое и часть создаются и разрушаются одновременно. Например, если между классами Window (Окно) и Button (Кнопка) установлена агрегация по значению, соответствующие объекты создаются и разрушаются в одно и то же время. На языке UML агрегацию по значению помечают закрашенным

ромбиком (рис. 2.27, а). ***Агрегация по ссылке*** (By Reference) предполагает, что целое и часть могут создаваться и разрушаться в разное время. Если между классом EmployeeList (Список сотрудников) и Employee (Сотрудник) установлена агрегация по ссылке, это означает, что они могут создаваться и разрушаться в памяти независимо друг от друга, то есть в разное время. Агрегация по ссылке изображается в виде пустого ромбика (рис.2.27, б).

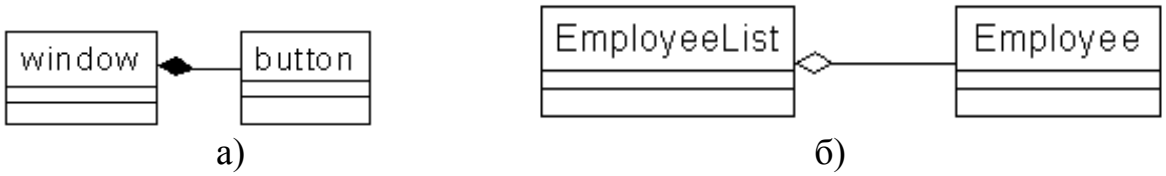


Рис. 2.27. Графическая нотация для изображения:
а) агрегации по значению; б) агрегации по ссылке

Агрегация не требует обязательного физического включения ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть косвенной. Например, объект класса Shareholder (Акционер) может содержать ключ записи об этом акционере в базе данных акций. Это тоже агрегация, однако, без физического включения.

«Лакмусовая бумажка» для выявления агрегации такова: если налицо отношение «целое/часть» между объектами, их классы должны находиться друг с другом в отношении агрегации. Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Если Вы не уверены, что налицо отношение общего/частного («is-a»), лучше вместо наследования применить агрегацию или что-нибудь еще.

4. С помощью ***обобщений*** (generalization) показывают связи наследования между двумя классами. ***Наследование*** - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование). Наследование позволяет одному классу наследовать все атрибуты, операции и связи другого. Класс, структура и поведение которого наследуются, называется ***суперклассом*** (предком). Производный от суперкласса класс называется ***подклассом*** (потомком). Наследование устанавливает между классами иерархию «общего/частного». В подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Подкласс обычно расширяет или ограничивает существующую структуру

и поведение своего суперкласса. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.

Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. Хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем.

Классы, экземпляры которых не создаются, называются ***абстрактными***. Ожидается, что подклассы абстрактных классов (такие классы называют конкретными классами или листьями иерархического дерева) доопределят их до жизнеспособной абстракции, наполняя класс содержанием. В C++ существует возможность объявлять функции виртуальными. Метод объявленный виртуальным может быть в подклассе переопределен, а остальные - нет. Если виртуальные методы не переопределены, экземпляр такого класса невозможно создать.

Большинство объектно-ориентированных языков непосредственно поддерживает концепцию наследования.

Наследования изображают в виде стрелки от класса-потомка к классу-предку (рис. 2.28).

Рис. 2.28. Связь обобщения

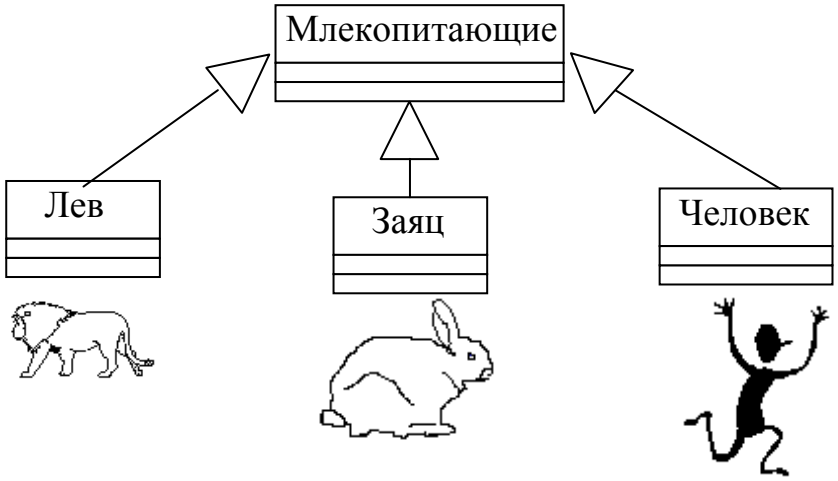


Рис. 2.29. Наследование: класс Млекопитающие

Примеры наследования можно обнаружить в природе. Существуют сотни различных типов млекопитающих: люди, львы, зайцы и т.д. У каждого из них имеются уникальные особенности, но есть и общие характеристики, такие, как принадлежность к теплокровным и воспитание потомства. Класс Млекопитающие имеет общие характеристики. Он порождает подклассы: Человек, Лев, Заяц и другие (рис.2.29). Подкласс наследует характеристики класса Млекопитающие, но в то же время имеет и свои собственные. Когда происходит событие, влияющее на всех млекопитающих, изменение надо внести только в родительский объект, а все потомки автоматически его наследуют. Если млекопитающие, например, внезапно станут холоднокровными, достаточно будет изменить класс Млекопитающие, подклассы Человек, Лев, Заяц и все другие автоматически наследуют новую характеристику.

В банковской системе наследование можно применять для работы с различными типами счетов. Допустим, банк обслуживает четыре типа счетов: до востребования (checking), сберегательный (savings), кредитный (credit) и депозитный сертификат. Все счета имеют следующие сходные характеристики: номер счёта, ставка процента и владелец. Можно создать родительский класс Account (Счёт) с общими характеристиками всех счетов. Подклассы наследуют характеристики суперкласса и имеют свои собственные уникальные характеристики. Например, кредитный счёт содержит лимит кредита и размер минимального взноса, а депозитный сертификат - срок платежа. Изменения в родительском классе повлияют на всех потомков, но потомки не влияют друг на друга и на своего предка.

Одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющийся код. Например, нельзя унаследовать графический элемент, который был бы одновременно окружностью и картинкой; приходится наследовать что-то одно и добавлять необходимое от второго. Механизм ***множественного наследования*** позволяет разрешить эти проблемы.

Если в структуре классов конечные классы (листья) могут быть сгруппированы во множества по разным ортогональным признакам, и эти множества перекрываются, тогда невозможно обойтись одной структурой наследования. В примере, описанном ниже, такими признаками являются способность приносить дивиденды и возможность страховки. В этой ситуации, чтобы соединить два нужных поведения, используют множественное наследование.

Рассмотрим пример множественного наследования. Допустим, что нужно организовать учет различных видов материального и нематериального имущества: банковские счета, недвижимость, акции и облигации. Банковские счета бывают текущие и сберегательные. Акции и облигации

можно отнести к ценным бумагам, управление ими отлично от банковских счетов. Но и счета и ценные бумаги - это разновидности имущества. Существуют другие классификации тех же видов имущества. Общим свойством банковских счетов и ценных бумаг является способность приносить дивиденды. Другим общим свойством является то, что их (недвижимость и сберегательные вклады) можно застраховать.

Одиночное наследование в данном случае не отражает реальности, поэтому будет использоваться множественное. Получившаяся структура классов показана на рис. 2.30. На нем класс Security (ценные бумаги) наследует одновременно от классов InterestBearingItem (источник дивидендов) и Asset (имущество). Аналогично, BankAccount (банковский счет) наследует сразу от трех классов: InsurableItem (страхуемое), Asset и InterestBearingItem.

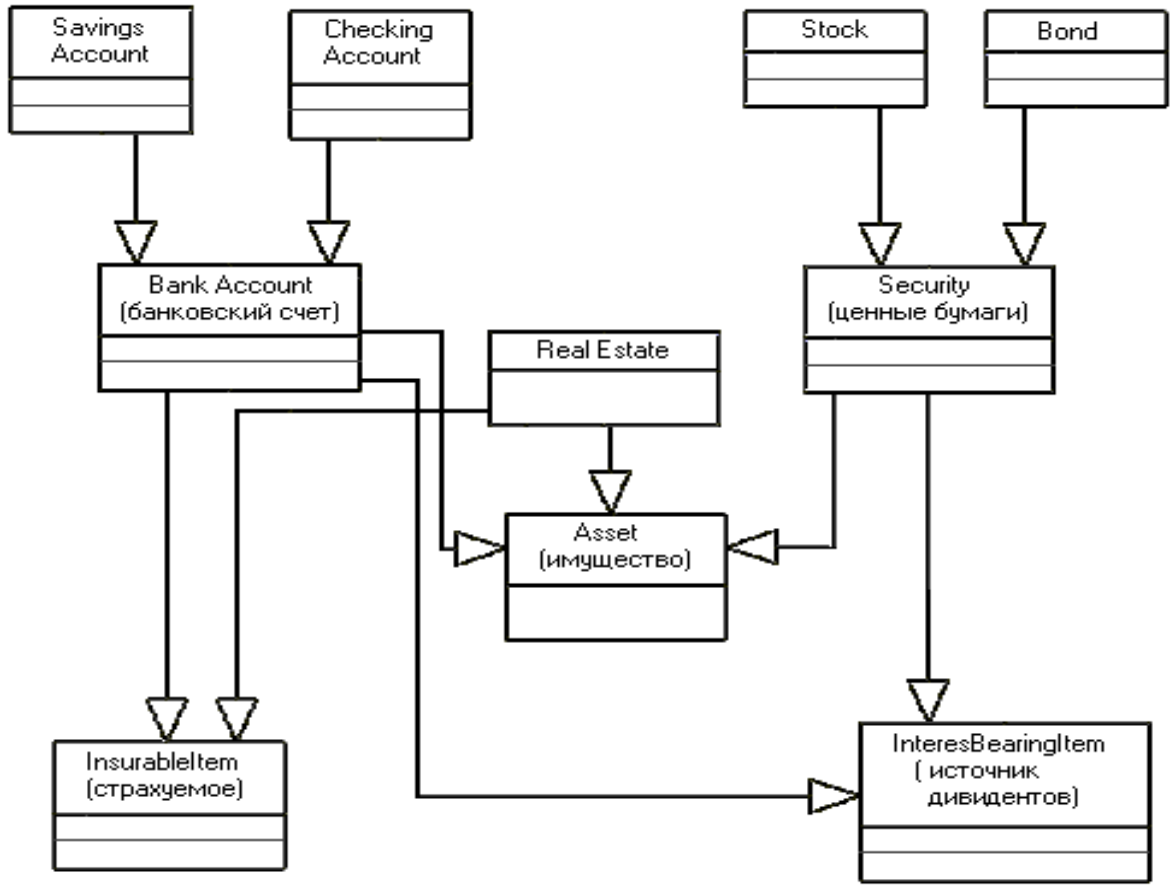


Рис. 2.30. Пример множественного наследования

Запишем, как это выражается на C++. Сначала базовые классы:

```
class Asset ...
class InsurableItem ...
class InterestBearingItem ...
```

Промежуточные классы (наследуют от нескольких суперклассов):

```
class BankAccount: public Asset,
                   public InsurableItem,
                   public InterestBearingItem ...
class RealEstate:  public Asset,
                   public InsurableItem ...
class Security:    public Asset,
                   public InterestBearingItem ...
// Листья:
class SavingsAccount: public BankAccount ...
class CheckingAccount: public BankAccount ...
class Stock: public Security ...
class Bond: public Security ...
```

Проектирование структур классов с множественным наследованием осуществляется путем последовательных приближений. Есть две специфические для множественного наследования проблемы: как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Допустим, классы Asset и InsurableItem содержат атрибут presentValue (текущая стоимость). Класс RealEstate потомок обоих классов, получается, что он наследует две операции с одним и тем же именем. Конфликт имен может ввести двусмысленность в поведение класса с несколькими предками. Для устранения этого к именам элементов добавляют префиксы, указывающие на имена классов-источников.

Вторая проблема множественного наследования возникает, когда один класс является наследником другого по нескольким линиям. Например, класс D наследует от B и C, которые, в свою очередь, наследуют от A. Эта ситуация называется ***повторным наследованием***. Рассмотрим следующий класс, который дважды наследует от класса Security:

```
class MutualFund: public Stock, public Bond ...
```

Проблема повторного наследования решается двумя способами. Во-первых, можно разделить две копии унаследованного элемента, добавив к именам префиксы в виде имени класса-источника. Во-вторых, можно рассматривать множественные ссылки на один и тот же класс как обозначающие один и тот же класс. Так поступают в C++, где повторяющийся суперкласс определяется как виртуальный базовый класс. Виртуальный базовый класс появляется, когда какой-либо подкласс именует другой класс своим суперклассом и отмечает этот суперкласс как виртуальный, чтобы показать, что это общий (shared) класс.

При множественном наследовании часто используют ***классы-примеси*** (mixin). В этом случае комбинируют (смешивают) небольшие классы, чтобы построить классы с более сложным поведением. Примесь синтаксически ничем не отличается от класса, но назначение их различно. Примесь не предназначена для порождения самостоятельно используемых экземпляров, она смешивается с другими классами. На рис. 2.30 классы InsurableItem и InterestBearingItem - это классы-примеси. Ни один из них не может существовать сам по себе, они используются для придания смысла другим классам. Таким образом, примесь - это класс, выражающий не поведение, а одно свойство, которое можно «привить» другим классам через наследование. Это свойство обычно ортогонально собственному поведению наследующего его класса. Классы, сконструированные только из примесей, называют агрегатными.

5. Наследование связано с явлением ***полиморфизма.***

Традиционные типизированные языки типа Pascal основаны на той идее, что функции и процедуры, а, следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом, то есть каждая переменная и каждое значение относятся к одному определенному типу. В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам.

Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Наследование без полиморфизма возможно, но не очень полезно. Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя.

Рассмотрим иерархию, в которой имеется базовый класс и три подкласса с именами Circle (Окружность), Triangle (Треугольник) и Rectangle (Прямоугольник) (рис. 2.31). Для класса Rectangle определен в свою очередь подкласс SolidRectangle (Окрашенный прямоугольник). Предположим, что в классе DisplayItem определена переменная экземпляра theCenter (координаты центра изображения), а также следующие операции: draw (нарисовать изображение), move (передвинуть изображение), location (вернуть координаты изображения). Операция location является общей для всех подклассов и не требует обязательного переопределения. Однако, поскольку только подклассы могут знать, как их изображать и передвигать, операции draw и move должны быть переопределены.

Класс Circle имеет переменную theRadius и, соответственно, операции для установки (set) и чтения значения этой переменной. Для этого

класса операция draw формирует изображение окружности заданного радиуса с центром в точке theCenter. В классе Rectangle есть переменные theHeight и theWidth и соответствующие операции установки и чтения их значений. Операция draw в данном случае формирует изображение прямоугольника заданной высоты и ширины с центром в заданной точке theCenter. Подкласс SolidRectangle наследует все особенности класса Rectangle, но операция draw в этом подклассе переопределена. Сначала вызывается draw вышестоящего класса, а затем полученный контур заполняется цветом.

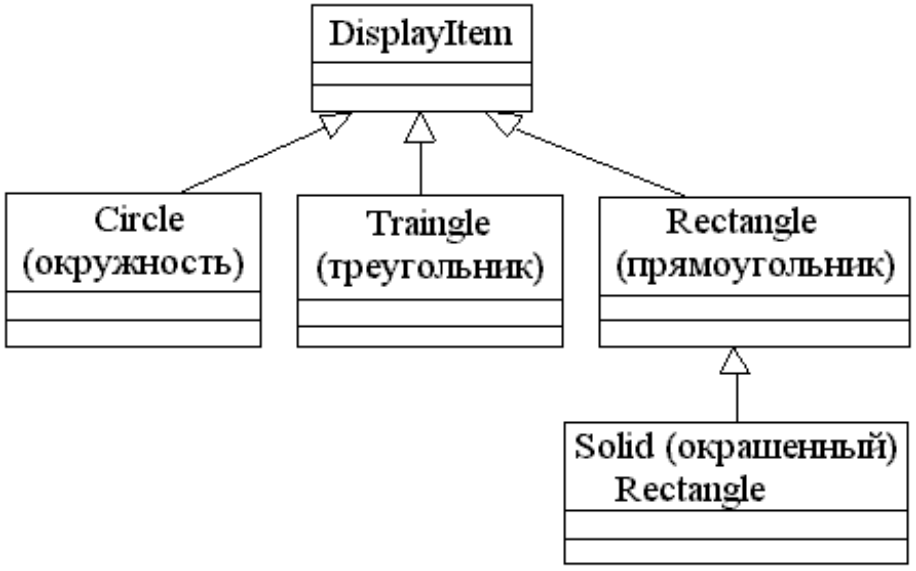


Рис. 2.31. Диаграмма класса DisplayItem

Рассмотрим следующий фрагмент программы:

```
DisplayItem* items[10]; ...
for (unsigned index=0;index<10;index++) items[index]->draw();
```

Вызов draw требует полиморфного поведения. Существует массив объектов, содержащий указатели на любые разновидности DisplayItem. Если нужно, чтобы все фигуры изобразили себя на экране, необходимо перебрать элементы массива и послать каждому указываемому объекту сообщение draw. Компилятор не может определить, какую функцию и откуда нужно при этом вызвать, так как невозможно предсказать, на что будут указывать элементы массива во время выполнения программы.

В C++ операции для позднего связывания объявляются виртуальными (virtual), а все остальные обрабатываются компилятором как обычные вызовы подпрограмм. В нашем примере draw - виртуальная функция, а location - обычная. Есть еще одно средство, используя которое можно выиграть в скорости. Невиртуальные методы могут быть объявлены подстав-

ляемыми (inline), при этом соответствующая подпрограмма не вызывается, а явно включается в код на манер макроподстановки.

Для управления виртуальными функциями в C++ используется концепция vtable (виртуальных таблиц), которые формируются для каждого объекта при его создании (то есть когда класс объекта уже известен). Такая таблица содержит список указателей на виртуальные функции. Например, при создании объекта класса Rectangle виртуальная таблица будет содержать запись для виртуальной функции draw, содержащую указатель на ближайшую в иерархии реализацию функции draw. Если в классе DisplayItem есть виртуальная функция rotate, которая в классе Rectangle не переопределена, то соответствующий указатель для rotate останется связан с классом DisplayItem. Во время исполнения программы происходит косвенное обращение через соответствующий указатель и сразу выполняется нужный код без всякого поиска.

Операция draw в классе SolidRectangle представляет собой особый случай. Чтобы вызвать метод draw из суперкласса, применяется специальный префикс, указывающий на место определения функции. Это выглядит следующим образом:

```
Rectangle::Draw();
```

Исследование Страуструпа показало, что вызов виртуальной функции по эффективности мало уступает вызову обычной функции [3]. Для одиночного наследования вызов виртуальной функции требует дополнительно выполнения трех-четырех операций доступа к памяти по отношению к обычному вызову; при множественном наследовании число таких дополнительных операций составляет пять или шесть.

Рассмотрим возможность ***множественного полиморфизма***. Вернемся к одной из функций-членов класса DisplayItem:

```
virtual void draw();
```

Эта операция изображает объект на экране в некотором контексте. Она объявлена виртуальной, то есть полиморфной, переопределяемой подклассами. Когда эту операцию вызывают для какого-то объекта, программа определяет, что, собственно, выполнять. Это одиночный полиморфизм в том плане, что смысл сообщения зависит только от одного параметра, а именно, объекта, для которого вызывается операция.

На самом деле операция draw должна бы зависеть от характеристик используемой системы отображения, в частности от графического режима. Например, в одном случае мы хотим получить изображение с высоким разрешением, а в другом - быстро получить черновое изображение. Можно ввести две различных операции, скажем, drawGraphic и drawText, но это не совсем то, что хотелось бы. Дело в том, что каждый раз, когда требуется

учесть новый вид устройства, его надо проводить по всей иерархии надклассов для класса DisplayItem.

В некоторых языках есть так называемые мультиметоды. Они полиморфны, то есть их смысл зависит от множества параметров (например, от графического режима и от объекта). В C++ мультиметодов нет, поэтому там используется идиома, так называемой, двойной диспетчеризации. Например, мы могли бы вести иерархию устройств отображения информации от базового класса DisplayDevice, а затем определить метод класса DisplayItem так:

```
virtual void draw(DisplayDevice&);
```

При реализации этого метода мы вызываем графические операции, которые полиморфны относительно переданного параметра типа DisplayItem. Таким образом, происходит двойная диспетчеризация: draw сначала демонстрирует полиморфное поведение в зависимости от того, к какому подклассу класса DisplayItem принадлежит объект, а затем полиморфизм проявляется в зависимости от того, к какому подклассу класса DisplayDevice принадлежит аргумент. Эту идиому можно продолжить до множественной диспетчеризации.

2.5.6.2. Дружественные связи

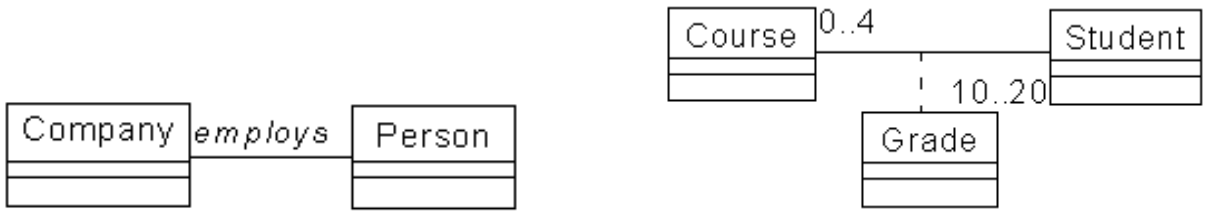
Дружественная (friend) связь предполагает, что класс-клиент имеет право доступа к атрибутам и операциям класса-сервера, которые не являются общими. Это свойство можно задать для ассоциаций, агрегаций, зависимостей и обобщений. В исходный код класса-сервера войдет логика, поддерживающая дружественную видимость для клиента. Допустим, что имеется двунаправленная ассоциация, связывающая классы Person и Company и между ними установлена дружественная связь. Тогда при генерации кода класс Person получит право доступа к частям класса Company, не являющимся общими.

2.5.6.3. Стереотипы, имена и элементы связей

Как и другим элементам модели, связям разрешается назначать ***стереотипы***. Они применяются для классификации связей. Например, Вы используете два типа ассоциаций, им можно задать стереотипы. Стереотипы пишут вдоль линии ассоциации в двойных угловых скобках (<< >>).

С помощью имен связей (ролевых имен) уточняют связи. ***Имя связи*** - это обычно глагол (глагольная фраза), описывающая, зачем нужна связь.

Например, между классом Person (Человек) и классом Company (Компания) существует ассоциация. Можно задать вопрос: «Зачем нужна эта связь? Является ли объект класса Person клиентом компании, ее сотрудником или владельцем?» Для прояснения ситуации можно назвать ассоциацию «employs» (нанимает). Имя показывают около линии соответствующей связи (рис. 2.32, а). Имена у связей определять необязательно. Обычно это делают, если причина создания связи неочевидна.



а) б)

Рис. 2.32. Графическая нотация для изображения:
а) имени связи; б) элемента связи (класса Ассоциаций)

Элементом связи (link element), известным также как класс Ассоциаций (Association class), называется место, где хранятся относящиеся к ассоциации атрибуты. Допустим, что имеются два класса: Student (Студент) и Course (Курс), и необходимо добавить на диаграмму атрибут Grade (Год обучения). В этом случае возникает вопрос, в какой класс добавить атрибут? Если мы поместим его в класс Student, то придется вводить атрибут для каждого посещаемого студентом курса, что значительно увеличит этот класс. Если же мы поместим его в класс Course, то придется задавать атрибут для каждого посещающего этот курс студента. Для решения этой проблемы можно создать класс Ассоциаций. В него следует поместить атрибут Grade, относящийся в большей степени к связи между классами Курс и Студент, чем к какому-то классу конкретно. Класс Ассоциаций изображается с помощью нотации показанной на рис. 2.32, б.

2.5.6.4. Выявление связей

Почти вся информация о связях находится на диаграммах Взаимодействия. Просмотрев их можно найти ассоциации и зависимости. Изучив имеющиеся классы можно обнаружить агрегации и обобщения. Для обнаружения связей рекомендуется выполнить следующие действия.

1. Начните с изучения диаграмм Взаимодействия. Если класс А посылает сообщение классу В, между этими классами может быть установлена связь. Как правило, обнаруживаемые таким способом связи - это ассоциации или зависимости.

2. Исследуйте классы на предмет наличия связей целое/часть. Любой класс, состоящий из других классов, может участвовать в связях агрегации.

3. Исследуйте классы на предмет связей обобщения. Ищите абстракции, которые бывают нескольких видов. Допустим, класс Employee (Сотрудник) содержит общие для всех сотрудников атрибуты, операции и связи. Однако в компании имеются два типа сотрудников: получающие почасовую оплату и оклад. Это означает, что можно создать классы HourlyEmp и SalariedEmp, наследующие от класса Employee, но содержащие элементы уникальные для сотрудников одного из типов.

4. Связи обобщения также можно обнаружить, исследуя классы, которые имеют много общего. Допустим, имеются классы CheckingAccount (Счет до востребования) и SavingsAccount (Сберегательный счет). Их данные и поведение сходны. Для общих элементов двух классов можно создать третий класс Account (Счет).

Будьте осторожны, работая с классами, у которых слишком много связей. Одной из особенностей хорошо спроектированного приложения является сравнительно небольшое количество связей в системе. Класс, у которого много связей, должен знать о большом числе других классов системы. В результате его трудно будет использовать во второй раз. Кроме того, сложно будет вносить изменения в готовое приложение. Если изменится любой из классов, это может повлиять на данный.

2.6. Качество классов и объектов

Классы и объекты являются ключевыми абстракциями системы. Опыт показывает, что процесс их выявления должен быть итеративным. За исключением самых простых задач, с первого раза не удается окончательно выделить и описать классы. Это влечет за собой дополнительные затраты на перекомпиляцию, согласование и внесение изменений в проект системы. Поэтому важно с самого начала максимально приблизиться к правильным решениям, чтобы сократить число последующих итераций. Для оценки качества классов и объектов используют пять критериев: сцепление, связность, достаточность, полнота, примитивность.

Термин «сцепление» взят из структурного проектирования. ***Сцепление*** является мерой внешней независимости между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения сцепления между отдельными модулями. При объектно-ориентированной разработке, кроме сцепления между модулями, существенную роль играет сцепление между классами и объектами. Существует определенное противоречие между сцеплением и наследованием. С одной стороны, желательно избегать сильного сцепления классов; с другой сто-

роны, механизм наследования, тесно связывающий подклассы с суперклассами, позволяет использовать сходство абстракций.

Понятие связности также заимствовано из структурного проектирования. ***Связность*** является мерой независимости внутренних частей модуля. При объектно-ориентированной разработке связность относится также к отдельным классам или объектам. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Примером может быть класс, объединяющий розы и космические аппараты. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля выполняют одну специфическую функцию. Эта функция может быть представлена набором элементарных функций, однако каждая из них не будет являться самостоятельной с учетом общей выполняемой работы. Так, например, класс Rose (Роза) будет функционально связным, если он описывает только поведение розы и ничего кроме этого.

К идеям сцепления и связности тесно примыкают понятия достаточности, полноты и примитивности.

Под ***достаточностью*** подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Компоненты должны быть полностью готовы к использованию. Допустим, в классе Set (множество) имеется операция удаления элемента из множества, будет ошибкой не включить в него также операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию.

Под ***полнотой*** подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями.

Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося наверх такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование примитивности. ***Примитивными*** являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом Set операция Add (добавление к множеству элемента) примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Примером примитивной операции также может служить операция добавления к множеству произвольного числа элементов (а не обязательно четырех).

2.7. Визуальное моделирование

2.7.1. Визуальное моделирование. Метод Буча, OMT и UML

Собираясь сделать пристройку к дому, Вы, вероятно, не начнёте с того, что купите кучу досок и будете сколачивать их вместе различными способами, пока не получите что-то приблизительно подходящее. Для работы понадобятся проекты и схемы, так что Вы скорее всего начнёте с планирования и структурирования пристройки. Тогда результат будет долговечнее, и дом не разрушится от небольшого дождика.

В мире программного обеспечения то же самое делается с помощью моделирования. Модели представляют собой проекты систем. Проект дома позволяет планировать его до начала непосредственной работы, ***модель*** даёт возможность спланировать систему до того, как вы приступите к её созданию. Это гарантирует, что проект удастся, требования будут учтены и система сможет выдержать «ураган» последующих изменений.

Визуальным моделированием называется процесс графического представления модели с помощью некоторого стандартного набора графических элементов. Наличие стандарта необходимо для реализации одного из преимуществ визуального моделирования - коммуникации. Общение между пользователями, разработчиками, аналитиками, тестировщиками, менеджерами и всеми остальными участниками проекта является основной целью визуального моделирования. Общение можно обеспечить с помощью невизуальной (текстовой) информации, но люди легче понимают сложную информацию, если она представлена визуально, нежели описана в тексте. Создавая визуальную модель системы, мы можем показать её работу на различных уровнях – можно моделировать взаимодействие между пользователем и системой, взаимодействие объектов внутри системы, а также взаимодействие между различными системами.

Созданные модели представляются всем заинтересованным сторонам, которые могут извлечь из них ценную информацию. Например, глядя на модель, пользователи визуализируют своё взаимодействие с системой. Аналитики увидят взаимодействие между объектами модели. Разработчики поймут, какие объекты нужно создать и что эти объекты должны делать. Тестировщики визуализируют взаимодействие между объектами, что позволит им построить тесты. Менеджеры увидят как всю систему в целом, так и взаимодействие её частей. Наконец, руководители информационной службы, глядя на высокоуровневые модели, поймут, как взаимодействуют друг с другом системы в их организации. Таким образом, визуальные модели предоставляют

мощный инструмент, позволяющий показать разрабатываемую систему всем заинтересованным сторонам.

Составление диаграмм - это еще не анализ и не проектирование. Диаграммы лишь помогают описать поведение системы (для анализа) или показать детали архитектуры (для проектирования). Будущая система сначала формируется в сознании разработчика, и только когда система будет понятна в общих чертах, она будет записана. Однако хорошо продуманная и выразительная система обозначений очень важна для разработки. Во-первых, общепринятая система позволяет разработчику описать сценарий или сформулировать архитектуру и доходчиво изложить свои решения коллегам. Символ транзистора понятен всем электронщикам мира. Аналогично, над проектом жилого дома, разработанным архитекторами в Санкт-Петербурге, строителям из Сан-Франциско скорее всего не придется долго ломать голову, решая, как надо расположить двери, окна или электрическую проводку. Во-вторых, как подметил Уайтхед в своей основополагающей работе по математике, «освобождая мозг от лишней работы, хорошая система обозначений позволяет ему сосредоточиться на задачах более высокого порядка» [3]. В-третьих, четкая система обозначений позволяет автоматизировать большую часть утомительной проверки на полноту и правильность. Кроме этого CASE-средства, используя принципы визуального моделирования, облегчают процесс разработки ПО. Они позволяют автоматизировать процесс создания и поддержания моделей в целостном состоянии, а также генерировать по разработанным моделям «скелетный» код системы.

Важный вопрос визуального моделирования – выбор графической нотации для описания различных аспектов системы. Нотация должна быть понятна всем заинтересованным сторонам, иначе модель будет бесполезна. Среди нотаций, которые используются объектно-ориентированной методологией, поддержку получили метод Буча, технология объектного моделирования (OMT, Object Modeling Technology) и унифицированный язык моделирования (UML, Unified Modeling Language). Однако большинством производителей и такими комитетами по стандартам, как ANSI и Object Management Group (OMG), был принят стандарт UML.

Метод Буча назван по имени его изобретателя Гради Буча (Grady Booch), работающего в корпорации Rational Software руководителем по науке (Chief Scientist). Он написал несколько книг, в которых обсуждаются необходимость и преимущества визуального моделирования, и разработал нотацию графических символов для описания различных аспектов модели. Объекты в его модели представляются в виде облаков, иллюстрируя то, что они могут быть почти чем угодно. Нотация Буча предусматривает также несколько видов стрелок для отображения разных типов отношений между объектами.

Нотация OMT была разработана Джеймсом Рамбо (Dr. James Rumbaugh), написавшим несколько книг о системном анализе и проектировании. В книге «System Analysis and Design» он рассматривает значимость моделирования систем с помощью компонентов (объектов) реального мира. Предложенная им нотация OMT получила широкое признание, ее поддерживают такие стандартные промышленные инструменты моделирования программного обеспечения, как Rational Rose и Select OMT. OMT использует более простую графику моделирования систем по сравнению с методом Буча.

Унифицированный язык моделирования (UML) является результатом совместных усилий Гради Буча, Джеймса Рамбо, Ивара Якобсона (Ivar Jacobson), Ребекки Вирс-Брок (Rebecca Wirfs-Brock), Питера Йордона (Peter Yourdon) и многих других. Якобсон первым описал процесс выявления и фиксации требований к системе в виде совокупностей транзакций, называемых вариантами использования (use case). Якобсон также разработал метод проектирования систем под названием «Объектно-ориентированное проектирование программного обеспечения» (Object Oriented Software Engineering, OOSE), концентрирующий внимание на анализе. Буч, Рамбо и Якобсон, о которых обычно говорят как о «трёх друзьях» (three amigos), работают в корпорации Rational Software. Их деятельность связана в основном со стандартизацией и усовершенствованием языка UML. Символы UML сильно напоминают нотации Буча и OMT, но содержат также элементы из других нотаций.

Процесс консолидации методов, вошедших в состав UML, начался в 1993 г. Каждый из трёх авторов этого языка – Буч, Рамбо и Якобсон – начал внедрять в свои разработки идеи остальных двух авторов. Такая унификация методологий продолжалась до 1995 г., когда появилась версия 0.8 Унифицированного Метода (Unified Method). В 1996 г. Унифицированный Метод был ратифицирован и передан в группу Object Technology Group, после чего его начали адаптировать у себя многие основные производители программного обеспечения. Наконец, 14 ноября 1997 г. группа OMG объявила язык UML 1.1 промышленным стандартом.

UML позволяет создавать несколько типов визуальных диаграмм, которые иллюстрируют различные аспекты системы:

1) Диаграммы Вариантов Использования;
2) Диаграммы Последовательности;
3) Кооперативные диаграммы;
4) Диаграммы Классов;
5) Диаграммы Состояний;
6) Диаграммы Компонентов;
7) Диаграммы Размещения.

Далее диаграммы UML рассматриваются подробнее.

2.7.2. Диаграммы Вариантов Использования

Диаграмма Вариантов Использования позволяет наглядно представлять требования к системе. Этот тип диаграмм описывает общую функциональность системы. Все участники проекта, изучая диаграммы Вариантов Использования, могут понять, что система должна делать. Диаграмма Вариантов Использования содержит варианты использования системы, действующих лиц и связи между ними. ***Вариант использования*** (use case) - это функции, выполняемые системой. ***Действующее лицо*** (actor) - это все, что взаимодействует системой (люди или системы, получающие или передающие информацию в данную систему). ***Связи*** позволяют показать способы взаимодействия между элементами системы. На рис. 2.33 приведен пример диаграммы Вариантов Использования автоматического банкомата (ATM). На ней показаны три действующих лица: клиент, банковский служащий и кредитная система. Существуют также шесть основных действий, выполняемых моделируемой системой: перевести деньги, положить деньги на счет, снять деньги со счета, показать баланс, изменить идентификационный номер и произвести оплату.

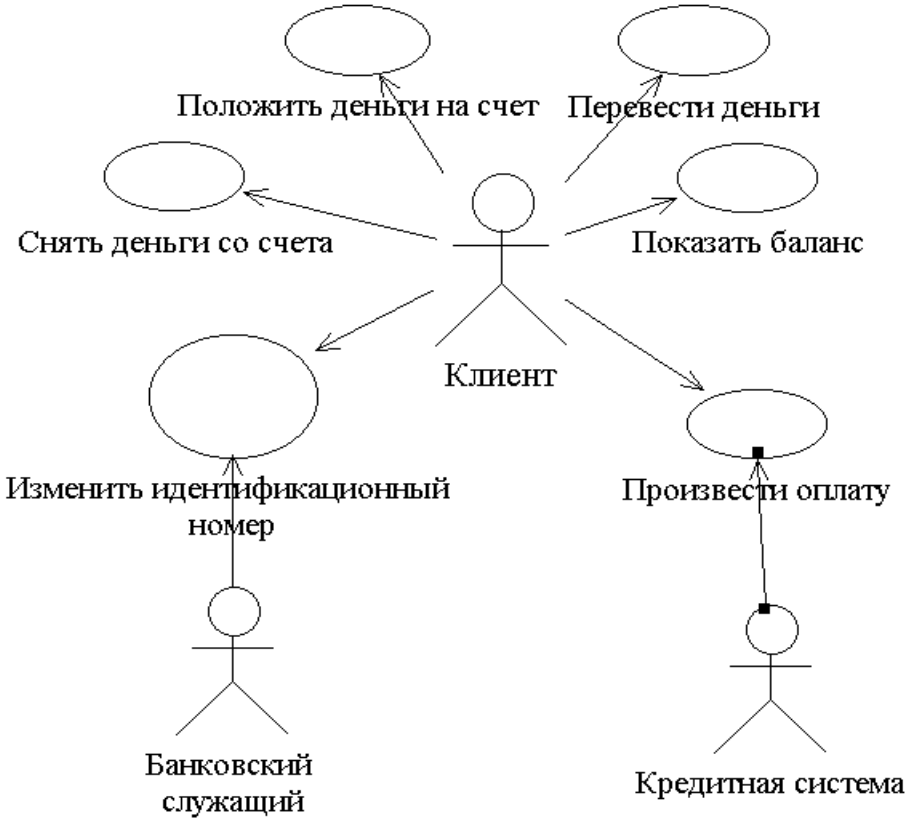


Рис. 2.33. Пример диаграммы Вариантов Использования

Из вариантов использования можно понять, какие функциональные возможности будут заложены в систему. Рассматривая действующих лиц можно выяснить, кто будет взаимодействовать с ней. Изучая все множество вариантов использования и действующих лиц, можно определить сферу применения системы, что она должна будет делать, а также что она не позволяет делать, и внести коррективы. Например, взглянув на приведенную в примере диаграмму, пользователь может сказать: «Я хочу дополнительно иметь возможность получать отчет о десяти последних транзакциях для моего счета».

Обычно для системы создается несколько диаграмм Вариантов Использования. На диаграмме высокого уровня (называемой «Главной» (Main)) указываются только группы (пакеты) вариантов использования. На других диаграммах эти группы конкретизируются, на них описываются варианты использования и действующие лица в рамках конкретной группы. Иногда может потребоваться нанести на одну диаграмму все варианты использования и всех действующих лиц системы. Количество и состав создаваемых диаграмм Вариантов Использования полностью зависит от разработчика. Важно только, чтобы они содержали достаточно информации, чтобы быть полезными, но не слишком много, чтобы не привести в замешательство.

Подробнее рассмотрим основные элементы диаграммы Вариантов Использования.

Действующее лицо (actor) - это то, что взаимодействует с создаваемой системой. Если варианты использования описывают все, что происходит внутри области действия системы, действующие лица определяют все, что находится вне этой области. На языке UML действующие лица представляются в виде фигур (рис. 2.33). Действующие лица делятся на три типа: пользователи системы, другие системы, взаимодействующие с данной, и время.

Первый тип действующих лиц - это ***физические личности***. Они наиболее типичны и имеются практически в каждой системе. В системе ATM, например, к этому типу относятся клиенты и обслуживающий персонал. Называя действующих лиц, используют их ролевые имена. Конкретный человек может играть множество ролей. Например, один человек утром может отвечать за поддержку системы ATM, т.е. относится к обслуживающему персоналу, а днем он может снять деньги со счета, чтобы пойти пообедать, при этом он уже является клиентом.

Второй тип действующих лиц – это ***другие системы***. Допустим, что у банка имеется кредитная система, используемая для работы с информацией о кредитных счетах клиентов. Если система ATM должна иметь возможность взаимодействовать с кредитной системой, в таком случае последняя становится действующим лицом. Нужно иметь в виду, что, делая систему действующим лицом, мы предполагаем, что она никогда не будет изменяться.

Действующие лица находятся вне сферы действия того, что мы разрабатываем, и, следовательно, не подлежат контролю с нашей стороны. Если мы собираемся изменять или разрабатывать и кредитную систему, то она попадает в наш проект и, таким образом, не может быть показана как действующее лицо.

Третий наиболее распространенный тип действующих лиц - ***время***. Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе. Например, система АТМ может каждую полночь выполнять какие-либо служебные процедуры по настройке к согласованию своей работы. Так как время не подлежит нашему контролю, оно является действующим лицом.

Абстрактным называется действующее лицо, не имеющее экземпляров. Например, в системе может быть несколько действующих лиц: служащий с почасовой оплатой, служащий с окладом и служащий, нанятый на определенное время. Все они являются разновидностями одного действующего лица – служащего (рис. 2.34). Однако никто в компании не является просто служащим - каждый относится к одному из трех вышеназванных типов. Действующее лицо «Служащий» существует только для того, чтобы показать общность между этими тремя типами. У него нет экземпляров, так что это абстрактное действующее лицо.

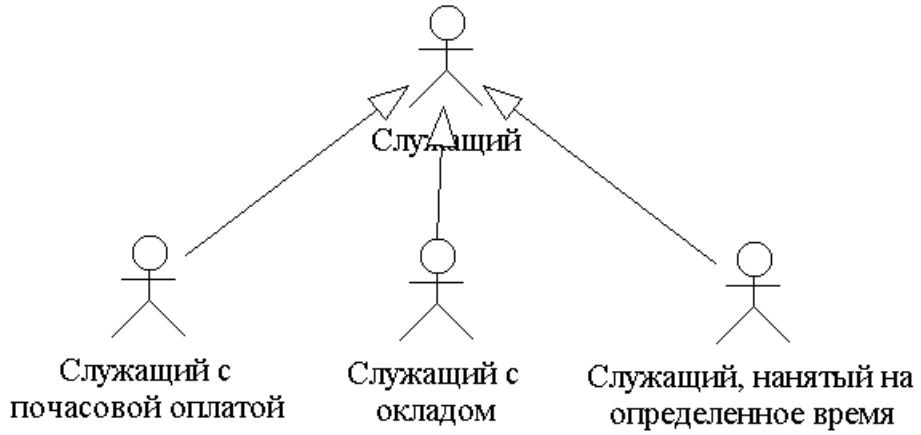
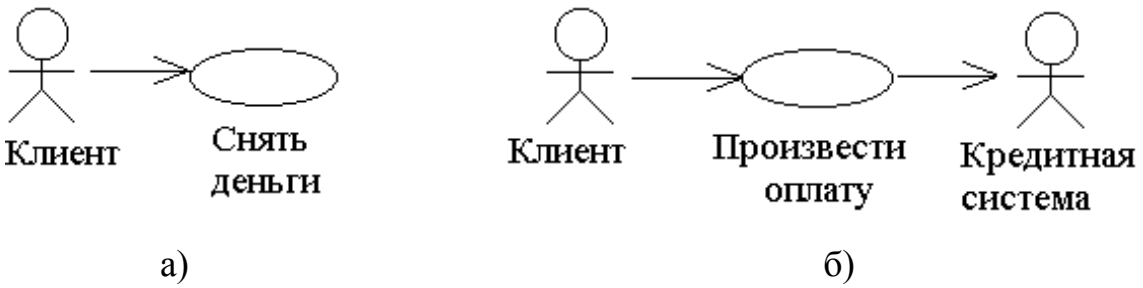


Рис. 2.34. Абстрактное действующее лицо

В языке UML для вариантов использования и действующих лиц поддерживается несколько типов ***связей***. Это связи: коммуникации (communication), использования (uses), расширения (extends) и обобщения действующего лица (actor generalization). Связи коммуникации описывают связи между действующими лицами и вариантами использования. Связи использования и расширения отражают связи между вариантами использова-

ния, а связи обобщения действующего лица - между действующими лицами.

Связь коммуникации (communicates relationship) - это связь между вариантом использования и действующим лицом. Связь коммуникации изображают в виде стрелки (рис. 2.35, а). Направление стрелки показывает, кто инициирует коммуникацию. В приведенном примере действующее лицо Клиент инициирует коммуникацию с системой для запуска функции «Снять деньги». Вариант использования также может инициировать коммуникацию с действующим лицом (рис. 2.35, б). В данном примере вариант использования инициирует коммуникацию с действующим лицом Кредитная система. Когда выполняется вариант использования «Произвести оплату», система АТМ инициирует коммуникацию с Кредитной системой, чтобы завершить транзакцию. Информация при этом движется в обоих направлениях: от АТМ к Кредитной системе и обратно; стрелка показывает, кто инициирует коммуникацию.



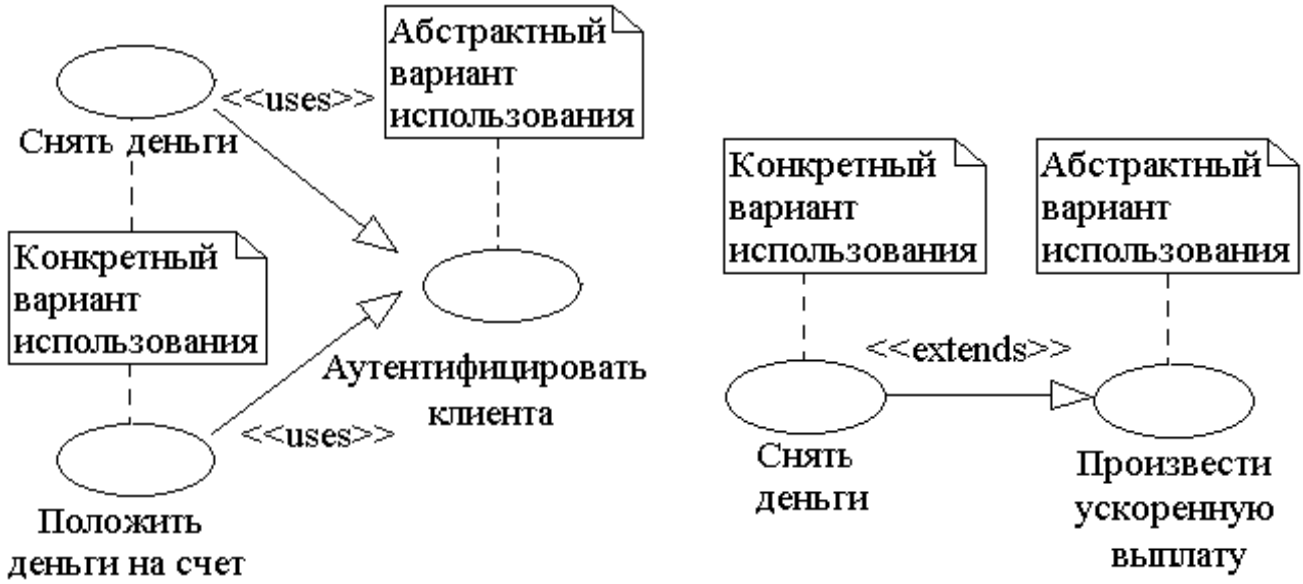
а) б)

Рис. 2.35. Графическая нотация для изображения:
а) однонаправленная коммуникация; б) вариант использования инициирует коммуникацию

Каждый вариант использования должен быть инициирован действующим лицом; исключения составляют лишь варианты использования в связях использования и расширения.

Связь использования (uses relationship) позволяет одному варианту использования задействовать функциональность другого. С помощью таких связей обычно моделируют многократно применяемую функциональность, встречающуюся в двух или более вариантах использования. В примере АТМ варианты использования «Снять деньги» и «Положить деньги на счет» должны опознать (аутентифицировать) клиента и его идентификационный номер перед тем, как разрешить выполнение 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». Когда какому-нибудь варианту использования потребуется выполнить эти действия, он сможет воспользоваться функциональностью созданного варианта использования «Аутентифи-

цировать клиента». Связь использования изображается с помощью стрелок и слова «uses» (использование), как показано на рис. 2.36, а.



а) б)

Рис. 2.36. Графическая нотация для изображения:
а) связи использования; б) связи расширения

Связь использования предполагает, что один вариант использования всегда применяет функциональные возможности другого. Например, независимо от того, как именно осуществляется вариант использования «Снять деньги», вариант использования «Аутентифицировать клиента» будет запущен в любом случае. ***Связи расширения*** (extends relationship) в отличие от использования позволяют варианту использования применять функциональные возможности, предоставляемые другим вариантом использования, только при необходимости. В связях расширения общая функциональность также выделяется в отдельный вариант использования.

Связи расширения изображают в виде стрелки со словом «extends» (расширение) (рис. 2.36, б). В этом примере вариант использования «Снять деньги» иногда применяет функциональные возможности, предоставляемые вариантом использования «Произвести ускоренную выплату». Это произойдет, например, когда клиент выберет пункт «Быстро снять $40». Предоставляющий дополнительные возможности вариант использования «Произвести ускоренную выплату» является абстрактным. Вариант использования «Снять деньги» - конкретный.

С помощью ***связи обобщения*** действующего лица (actor generalization

relationship) показывают, общие черты нескольких действующих лиц. Это отношение моделируется с помощью нотации, представленной на рис. 2.37, а. На диаграмме показано, что имеются два типа клиентов: индивидуальные и корпоративные. Это конкретные действующие лица. Они будут инстанцированы (наполнены) непосредственно. Действующее лицо Клиент - абстрактное. Оно никогда не инстанцируется непосредственно. Оно нужно лишь для того, чтобы показать наличие двух типов клиентов. При необходимости можно усиливать ветвление. Допустим, имеются два типа корпоративных клиентов: правительственные агентства и частные компании (рис. 2.37, б).

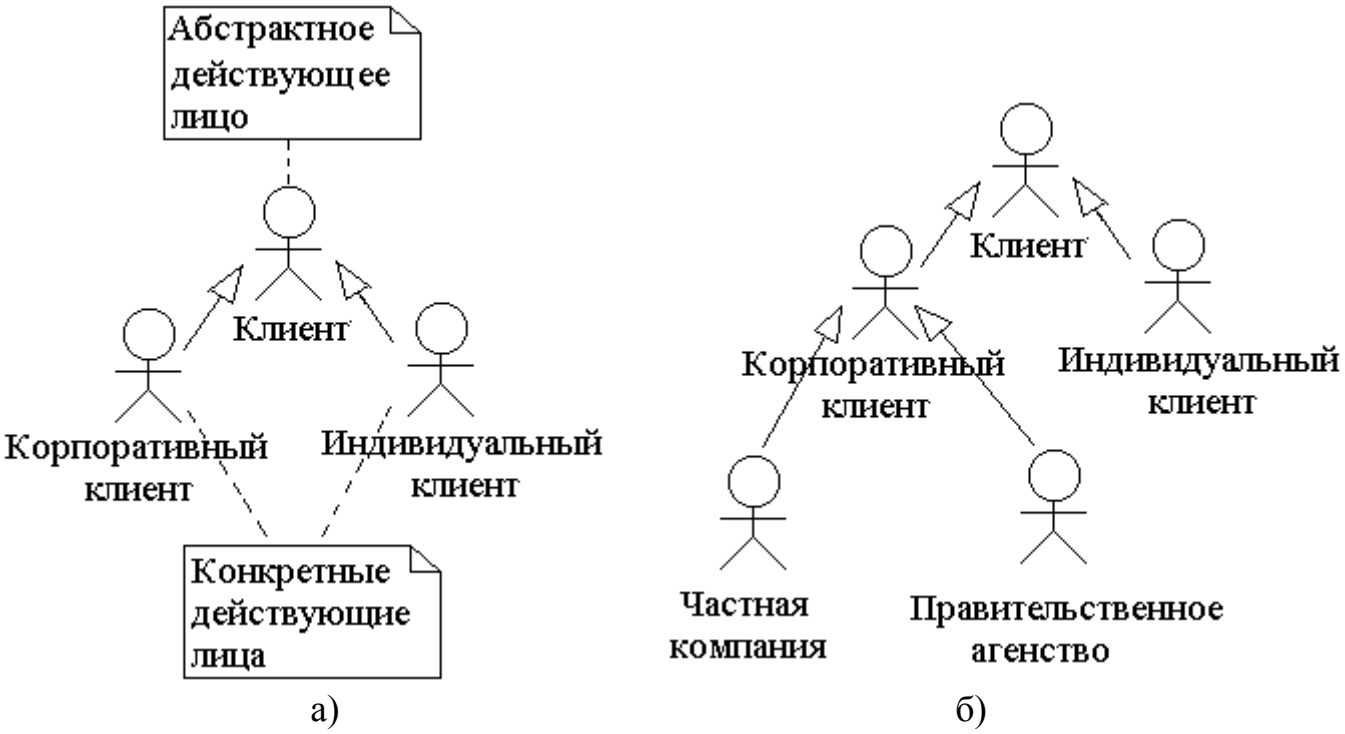


Рис. 2.37. Графическая нотация для изображения:
а) связи обобщения действующего лица; б) усиление ветвления связи обобщения

Связи обобщения создаются не всегда. В общем случае они нужны, если поведение действующего лица одного типа отличается от поведения другого настолько, что это влияет на систему. Если, например, корпоративные клиенты инициируют некоторые варианты использования, которые не могут быть запущены индивидуальными клиентами, то связи обобщения имеет смысл показать на диаграмме. Если клиенты обоих типов применяют одни и те же варианты использования, этого не требуется. Если действующие лица задействуют одни и те же варианты использования, но с некоторым отличием, их также не стоит включать в обобщение. Небольшие различия можно до-

кументировать в потоке событий варианта использования.

Описав основных элементов диаграмм Вариантов Использования, рассмотрим процесс их выявления.

В начале работы над проектом возникает естественный вопрос: «Как обнаружить варианты использования?». Можно обратится к документации заказчика, рассмотреть области использования системы на высоком уровне или документы концептуального характера. Нужно стараться учесть мнение каждого из заинтересованных лиц проекта и определить, что они ожидают от готового продукта. Каждому из них можно задать следующие вопросы.

1. Что он хочет делать с системой?
2. Будет ли он с ее помощью работать с информацией (вводить, получать, обновлять, удалять)?
3. Нужно ли будет информировать систему о каких-либо внешних событиях?
4. Должна ли система в свою очередь информировать пользователя о каких-либо изменениях или событиях?

Как уже упоминалось ранее, варианты использования - это не зависящее от реализации высокоуровневое представление того, что пользователь ожидает от системы. Рассмотрим каждый фрагмент этого определения.

Прежде всего, варианты использования не зависят от реализации. Нужно составлять варианты использования так, чтобы их можно было реализовать на любых языках (Java, C++, Visual Basic и т.п.). Варианты использования заостряют внимание на том, что должна делать система, а не на том, как она должна это делать.

Варианты использования - это высокоуровневое представление системы. Если, например, в вашей модели содержится 3000 вариантов использования, Вы теряете преимущество простоты. Создаваемый набор вариантов использования должен предоставить пользователям возможность увидеть всю систему целиком на самом высоком уровне. Поэтому вариантов использования не должно быть слишком много, чтобы клиенту не пришлось долго блуждать по страницам документации с целью выяснения того, что будет делать система. В то же время вариантов использования должно быть достаточно для полного описания поведения системы. Модель типичной системы содержит от 20 до 50 вариантов использования.

Наконец, варианты использования заостряют внимание на том, что пользователи хотят получить от системы. Каждый вариант использования должен представлять собой завершенную транзакцию между пользователем и системой. Названия вариантов использования должны быть деловыми, а не техническими терминами. Если в примере с АТМ назвать вариант использования «Интерфейс с банковской системой, осуществляющий перевод денег с

кредитной карточки и наоборот», это будет не приемлемо. Лучше дать название «Оплатить по карточке» - так будет понятнее для заказчика. Варианты использования обычно называют глаголами или глагольными фразами, описывая при этом, что пользователь видит как конечный результат процесса. Его не интересует, сколько других систем задействовано в варианте использования, какие конкретные шаги надо предпринять и сколько строчек кода требуется написать, чтобы заплатить по счету карточкой Visa. Для него важно только, чтобы оплата была произведена. Нужно заострять внимание на результате, который потребитель ожидает от системы, а не на действиях, которые нужно предпринять для достижения этого результата.

Как убедиться, что обнаружены все варианты использования? Для этого следует ответить на следующие вопросы.

1. Присутствует ли каждое функциональное требование хотя бы в одном варианте использования? Если требование не нашло отражение в варианте использования, оно не будет реализовано.
2. Учтено ли, как с системой будет работать каждое заинтересованное лицо?
3. Какую информацию будет передавать системе каждое заинтересованное лицо?
4. Какую информацию будет получать от системы каждое заинтересованное лицо?
5. Учтены ли проблемы, связанные с эксплуатацией? Кто-то должен будет запускать готовую систему и выключать ее.
6. Учтены ли все внешние системы, с которыми будет взаимодействовать данная?
7. Какой информацией каждая внешняя система будет обмениваться с данной?

Приведем пример создания диаграммы Вариантов Использования для системы обработки заказов. Система должна обеспечивать возможность добавления новых заказов, изменения старых, выполнения заказов, проверки и возобновления инвентарных описей. При получении заказа система должна послать сообщение бухгалтерской системе, которая выписывает счет. Если требуемого товара нет на складе, заказ должен быть отклонен.

Основными вариантами использования системы будут следующие: Ввести новый заказ, Изменить существующий заказ, Напечатать инвентарную опись, Обновить инвентарную опись, Оформить заказ, Отклонить заказ.

После выявления вариантов использования рассматривают возможные роли в системе и определяют действующих лиц. В данном примере будут следующие действующие лица: Продавец, Управляющий магазином, Клерк магазина, Бухгалтерская система.

На следующем шаге устанавливаются связи между выявленными элементами системы. За исключением одного случая, все связи однонаправленные ассоциации. Так как вариант использования «Отклонить заказ» при необходимости дополняет функциональные возможности варианта использования «Оформить заказ» между ними будет связь расширения.

Полученная диаграмма Вариантов Использования показана на рис. 2.38.

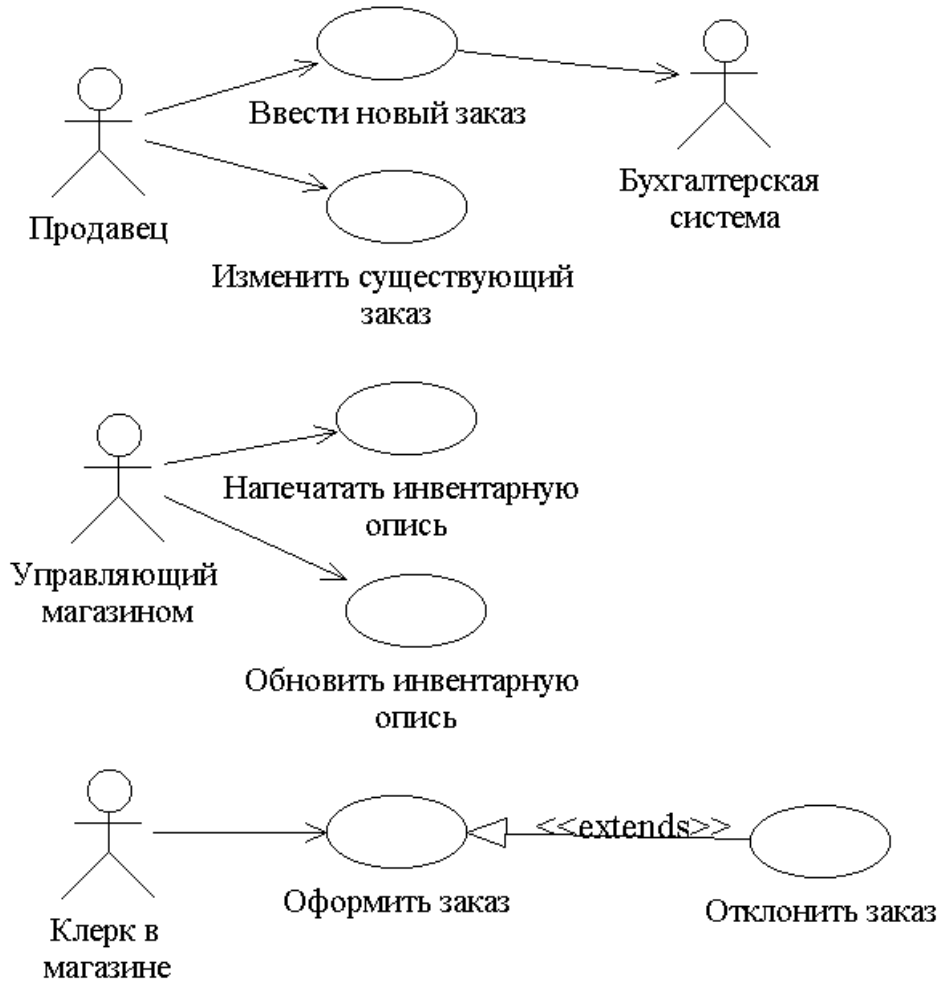


Рис. 2.38. Диаграмма вариантов использования системы обработки заказов

2.7.3. Потоки событий

Варианты использования начинают описывать, что должна будет делать система. Но чтобы фактически разработать систему, потребуются более конкретные детали. Они определяются в документе, называемом ***потоком со-***

бытий (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 сама система.

Поток событий не должен зависеть от реализации. Составляя этот документ, можно не учитывать на каком языке будет написана система (C++, PowerBuilder, Java или другом). Поток событий уделяет внимание тому, что (а не как) будет делать система. Обычно поток событий содержит:

1) краткое описание;
2) предусловия (pre-conditions);
3) основной поток событий;
4) альтернативный поток событий;
5) постусловия (post-conditions).

Рассмотрим эти составные части подробнее.

Каждый вариант использования должен иметь связанное с ним ***краткое 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».* Следует делать описание коротким и «к месту», при этом оно должно определять типы пользователей, выполняющих вариант использования, и ожидаемый ими конечный результат. Во время работы над проектом (особенно, если проект длинный) эти описания будут напоминать членам команды, почему тот или иной вариант использования был включен в проект и что он должен делать. Четко документируя цели каждого варианта использования, можно избежать противоречий среди разработчиков.

Предусловия варианта использования - это такие условия, которые должны быть выполнены, прежде чем вариант использования начнет свою работу. Например, таким условием может быть выполнение другого варианта использования или наличие у пользователя прав доступа, требуемых для запуска данного варианта использования. Не у всех вариантов использования бывают предварительные условия. Ранее упоминалось, что диаграммы Вариантов Использования не должны отражать порядок их выполнения. Однако с помощью предусловий можно документировать и такую информацию.

Конкретные детали вариантов использования отражаются в ***основном*** и ***альтернативном потоках событий***. Поток событий поэтапно описывает, что должно происходить во время выполнения заложенной в вариант использования функциональности. Поток событий уделяет внимание тому, что (а не как) будет делать система, причем описывает это с точки зрения пользователя. Первичный и альтернативный потоки событий содержат:

1) описание того, каким образом запускается вариант использования;
2) различные пути выполнения варианта использования;
3) нормальный, или основной, поток событий варианта использования;
4) отклонения от основного потока событий (так называемые альтернативные потоки);
5) потоки ошибок;
6) описание того, каким образом завершается вариант использования.

В качестве примера приведем потоки событий для варианта использования «Снять деньги» системы АТМ.

Основной поток.

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

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

3. Клиент вводит номер.

4. АТМ подтверждает введенный номер. Если номер не подтверждается, 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 нужно снять.

8. Клиент вводит требуемую сумму.

9. АТМ определяет, достаточно ли на счету 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. Вариант использования завершается.

Альтернативный поток А1: ввод неправильного идентификационного номера.

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

2. АТМ возвращает клиенту его карточку.

3. Вариант использования завершается.

Альтернативный поток 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 E1: 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.

Документируя поток событий, можно использовать нумерованные списки (как это сделано в данном примере), ненумерованные списки, разбитый на параграфы текст, а также блок-схемы. Поток событий должен быть согласован с определенными ранее требованиями.

Описывая поток событий, нужно учитывать, что с этим документом будут работать все участники проекта. При его изучении заказчики будут проверять, соответствует ли он их ожиданиям, аналитики - соответствует ли он требованиям к системе. Менеджер проекта определяет из потока событий, что будет создано и делает оценку проекта.

Работая над потоком, нужно избегать детальных обсуждений того, как он будет реализован. Подобно кулинарному рецепту, в котором просто указывается: «Добавьте сахар», а не расписывается: «Откройте шкаф. Возьмите пакетик с сахаром. Возьмите стакан. Насыпьте сахар в стакан…». Аналогично, составляя поток событий, можно написать: «Проверить идентификационный номер пользователя», но не нужно указывать, что для этого необходимо обращаться к специальной таблице в базе данных. Нужно уделять внимание обмену информацией между пользователями и системой, но не подробному описанию ее реализации.

Постусловиями называются такие условия, которые должны быть выполнены после завершения варианта использования. Например, в конце варианта использования нужно установить флажок. Как и в случае предусловий, с помощью постусловий можно вводить сведения о порядке выполнения вариантов использования системы. Если после одного из вариантов использования должен всегда выполняться другой, это можно описать как постусловие. Такие условия имеются не у каждого варианта использования.

2.7.4. Диаграммы Взаимодействия

Диаграммы Взаимодействия (Interaction) используются для моделирования взаимодействия между объектами системы. С помощью них проектировщики и разработчики системы могут определить классы, которые нужно создать, связи между ними, а также операции и ответственности (responsibilities) каждого класса. Диаграммы Взаимодействия - краеугольный камень, на котором возводится оставшаяся часть проекта.

Диаграммы Взаимодействия визуализируют практически те же детали, которые были описаны в потоке событий, однако представляют их в форме, более удобной для разработчика. Главное здесь - объекты, которые должны быть созданы для реализации функциональных возможностей, заложенных в вариант использования. Вариант использования может иметь несколько альтернативных потоков, диаграмма Взаимодействия описывает только один из них. Например, в случае с АТМ можно создать несколько таких диаграмм. На одной показать, что происходит, когда все в порядке. На других ход событий, когда клиент ввел неправильный идентификационный номер, или денег на его счету меньше, чем он хочет снять, и т.д.

Для построения диаграмм Взаимодействия является важным понятие «сценарий». Под ***сценарием*** понимается поток событий для частного случая. У каждого потока событий существует обычно много сценариев. Диаграмма Взаимодействия отображает один из сценариев. В рассмотренном ранее примере поток событий варианта использования «Снять деньги» говорит о человеке, снимающем некоторую сумму денег со счета с помощью АТМ. Один из сценариев этого потока может быть следующим: Джо снимает со счета $20. Другой сценарий - Джейн хочет снять $20, но вводит неправильный идентификационный номер. Еще один - Боб хочет снять $20, но на его счету недостаточно денег.

Существует два типа диаграмм Взаимодействия - диаграммы Последовательности (Sequence) и Кооперативные диаграммы (Collaboration). Рассмотрим основные элементы этих диаграмм на примере сценария, показывающего, как клиент банка Джо снимает со счета $20 с помощью автоматического банкомата (АТМ). Процесс начинается, когда Джо вставляет карточку в устройство для ее чтения. Это устройство считывает номер карточки и выдает экрану АТМ команду инициализации. АТМ запрашивает у Джо его идентификационный номер. Джо вводит свой номер (1234), и АТМ открывает его счет. Идентификационный номер Джо подтверждается, и АТМ предлагает Джо выбрать транзакцию. Джо указывает «Снятие денег». АТМ запрашивает требуемую сумму. Джо вводит $20. АТМ удостоверяется, что на счету Джо имеется достаточно денег, и вычитает $20 из его счета. Затем выдается требу-

емая сумма, и Джо получает обратно свою карточку.

Оба типа диаграмм Взаимодействия содержат объекты и сообщения.

В качестве ***имени объекта*** можно использовать имена не только объектов, но и классов, а также того и другого одновременно.

Сообщение (message) - это связь между объектами, в которой один из них (клиент) требует от другого (сервера) выполнения каких-то действий. Например, форма может запросить у объекта Отчет напечатать ее. Сообщения могут быть рефлексивными, что соответствует обращению объекта к своей собственной операции. Для сообщений можно использовать обычные фразы на русском языке.

Помещая на диаграмму Взаимодействия сообщение, Вы таким образом назначаете ответственность получающему его объекту. Нужно следить за тем, чтобы объекты и их ответственности соответствовали друг другу. Например, в большинстве приложений экраны и формы не должны реализовывать никаких бизнес-процессов. С их помощью следует только вводить и просматривать информацию. Отделяя интерфейс от бизнес-логики, Вы создаете архитектуру, которая меньше подвержена влиянию изменений. Внесение изменений в бизнес-логику не затронет интерфейс и наоборот. Приведем еще один пример. Допустим, нужно распечатать отчет о балансах на всех счетах. Этим не должен заниматься объект Счет Джо, так как его ответственности должны быть связаны только со счетом Джо и его деньгами. За просмотр всех счетов и генерацию соответствующего отчета должен отвечать другой объект.

На ***диаграмме Последовательности*** взаимодействия упорядочены во времени. Они показывают логическую последовательность событий в сценарии. Читать диаграммы Последовательности нужно сверху вниз. На рис. 2.39 приведен пример диаграммы Последовательности для рассматриваемого сценария. Участвующие в потоке объекты нарисованы в прямоугольниках в верхней части диаграммы. В нашем примере имеются пять объектов: Джо, устройство для чтения карточки, экран АТМ, счет Джо и кассовый аппарат. Объект-действующее лицо Джо, инициирующий вариант использования, показан в верхнем левом углу диаграммы.

У каждого объекта имеется ***линия жизни*** (lifeline), изображаемая в виде вертикальной штриховой линии под объектом.

На диаграмме Последовательности ***сообщение*** изображают в виде стрелки, которая проводится между линиями жизни двух объектов (от клиента к серверу) или от линии жизни объекта к самой себе. Сообщения располагают в хронологическом порядке сверху вниз, поэтому их нумерация необязательна. В нашем примере сообщение 2 рефлексивное, оно демонстрирует, что устройство чтения карточки обращается к самому себе, чтобы прочитать номер карточки.

Глядя на диаграмму Последовательности, пользователи знакомятся со спецификой своей работы. Аналитики видят последовательность (поток) действий, разработчики – объекты, которые надо создать, и их операции. Специалисты по контролю качества поймут детали процесса и смогут разработать тесты для их проверки. Таким образом, диаграммы Последовательности полезны всем участникам проекта.

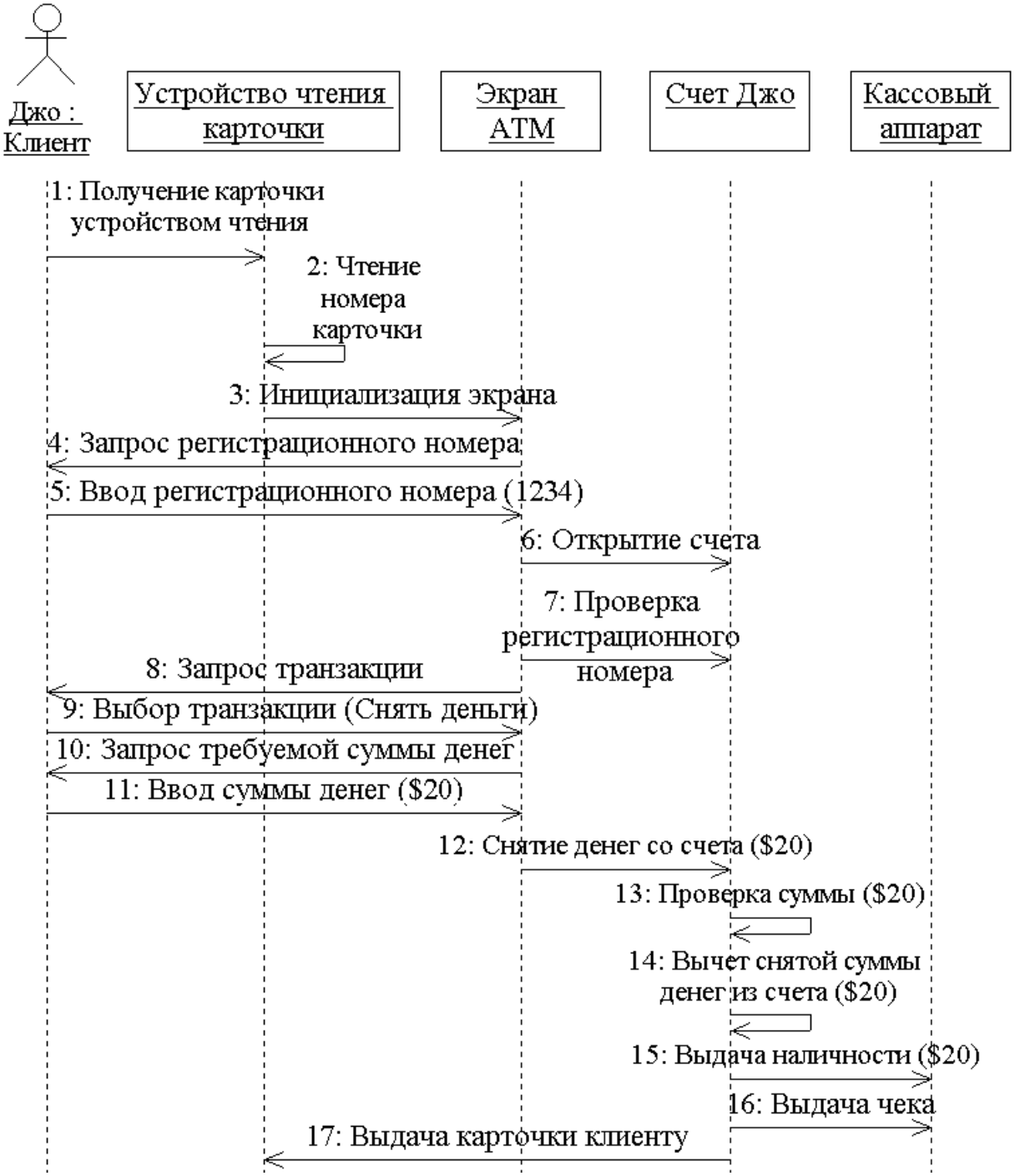


Рис. 2.39. Пример диаграммы Последовательности

Кооперативные диаграммы и диаграммы Последовательности содержат одну и ту же информацию, однако представляют ее с различных точек зрения. Если диаграмма Последовательности показывает взаимодействие между действующими лицами и объектами во времени, то на Кооперативной диаграмме связь со временем отсутствует, она больше внимания заостряет на связях между объектами. Кооперативные диаграммы полезны в тех случаях, когда нужно оценить последствия сделанных изменений. При внесении изменений в объект сразу становится ясно, на какие другие объекты это повлияет. Обычно для сценария создают диаграммы обоих типов. На рис. 2.40 приведена Кооперативная диаграмма, описывающая, как Джо снимает со счета $20.

Рис. 2.40. Пример Кооперативной диаграммы

На Кооперативной диаграмме связь между объектами или объекта с самим собой указывается в виде обычной линии, возле которой помещается текст сообщения. Сообщения нумеруются, иначе информация о порядке их

следования будет потеряна.

Специалисты по контролю качества и архитекторы системы из Кооперативной диаграммы смогут понять распределение процессов между объектами. Допустим, что какая-то Кооперативная диаграмма напоминает звезду, где несколько объектов связаны с одним центральным объектом. Архитектор системы может сделать вывод, что система слишком сильно зависит от центрального объекта, и перепроектировать ее для более равномерного распределения процессов. На диаграмме Последовательности такой тип взаимодействия было бы трудно увидеть.

Если на диаграмме Взаимодействия нужно отобразить список сотрудников, чтобы не представлять каждого сотрудника как отдельный объект, можно показать всех сотрудников сразу с помощью значка множественного экземпляра (рис. 2.41).

Рис. 2.41. Значок множественного экземпляра

Рассмотрим этапы составления диаграмм Взаимодействия.

1. Первым шагом является поиск объектов и действующих лиц. Один из способов первоначального выявления некоторых объектов - это изучение имен существительных в потоке событий и сценариях. Некоторые существительные в сценариях будут действующими лицами, другие - объектами, а третьи - атрибутами объекта. Если не ясно, что описывает существительное: объект или атрибут, подумайте, есть ли у него поведение. Если нет, то это, вероятно, атрибут. Если да, то, скорее всего, объект.

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 вводить новую информацию в систему или просматривать ее. В потоке событий, скорее всего не будет и управляющих объектов (control objects), которые управляют в варианте использования последовательностью потока.

2. После того как все объекты созданы, необходимо соотнести их с классами. Их можно связать с существующими классами или создать для них новые. В нотации UML к названию объекта соотнесенного с классом добавляется имя класса, указанное после двоеточия. Например, имя объекта Счет Джо соотнесенного с классом Account (Счет) будет «Счет Джо : Account».

3. На следующем шаге нужно добавить на диаграмму сообщения.

4. Далее каждое сообщение диаграммы Взаимодействия должно быть соотнесено с операцией. Теперь вместо сообщений на нее нужно поместить соответствующие имена операций. На рис. 2.48 приведен пример диаграммы Последовательности, на которую добавлены сообщения. На рис. 2.50 приведена та же диаграмма после соотнесения сообщений с операциями. Если диаграмма Последовательности выглядит так, как показано на рис. 2.50, то операция «1: Creat» будет расположена внутри класса Order.

5. На последнем шаге предстоит перейти к спецификациям самих объектов и сообщений и задать такие детали, как устойчивость объекта, синхронизация и частота сообщений.

Доступны пять значений параметра синхронизации: простое, синхронное, асинхронное, с отказом становиться в очередь, с лимитированным временем ожидания.

Simple (простое сообщение) используется по умолчанию. Оно означает, что все сообщения выполняются в одном потоке управления. В нотации UML простое сообщение изображается обычной стрелкой.

Synchronous (синхронное сообщение) применяется, когда клиент посылает сообщение и ждет ответа пользователя. Синхронные сообщения изображаются с помощью нотации показанной на рис. 2.42, а.

Клиент посылает ***asynchronous*** (асинхронное) сообщение серверу и продолжает свою работу, не ожидая подтверждения о получении. Асинхронные сообщения изображаются с помощью нотации показанной на рис. 2.42, б.

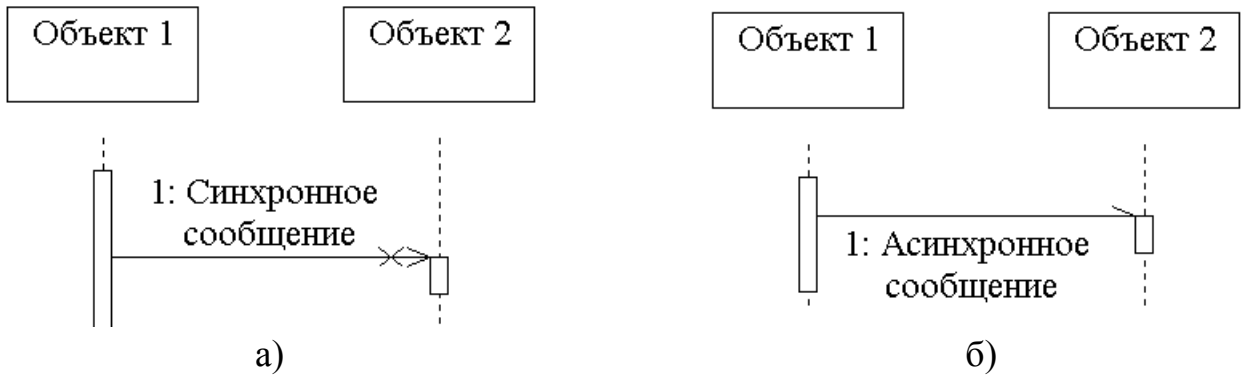


Рис. 2.42. Графическая нотация для изображения:
а) синхронного сообщения; б) асинхронного сообщения

Клиент посылает сообщение ***balking*** (сообщение с отказом становиться в очередь) серверу. Если сервер не может немедленно принять сообщение, оно отменяется. Сообщения с отказом становиться в очередь изображаются с

помощью нотации показанной на рис. 2.43, а.

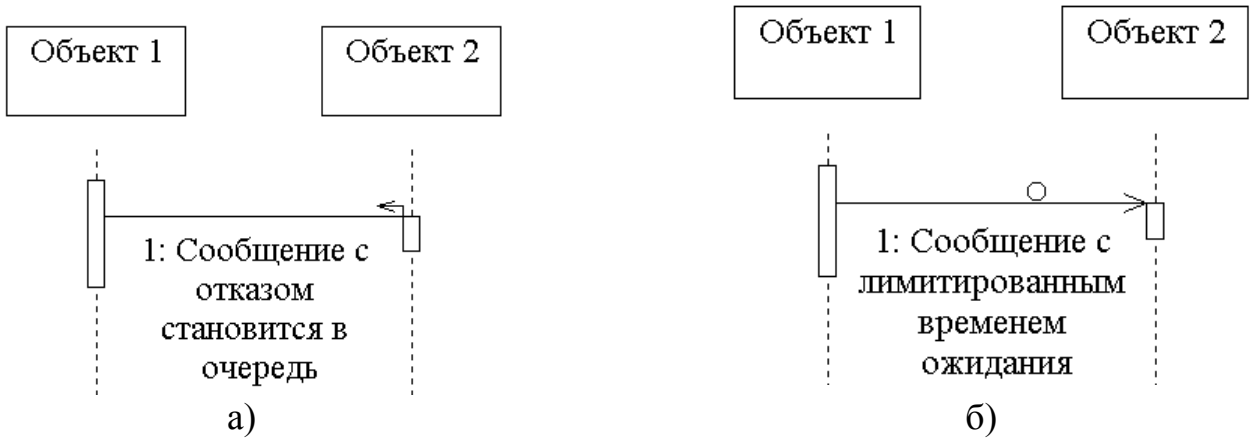


Рис. 2.43. Графическая нотация для изображения:
а) сообщения с отказом становиться в очередь; б) сообщения с лимитированным временем ожидания

Клиент посылает сообщение ***timeout*** (сообщения с лимитированным временем ожидания) серверу, а затем ждет указанное время. Если в течение этого времени сервер не принимает сообщение, оно отменяется. Сообщения такого типа изображаются с помощью нотации показанной на рис. 2.43, б.

Скрипты применяются только на диаграммах Последовательности и пишутся с левой стороны диаграммы напротив соответствующего сообщения.

С помощью скрипта можно пояснить назначение сообщения. Например, имеется сообщение «Проверка клиента». В скрипте этого сообщения можно разъяснить его значение следующим образом: «Проверить идентификационный номер, чтобы убедиться, что клиент существует, и что пароль введен правильно».

Кроме того, скрипты позволяют указывать на диаграмме некоторые условия (рис. 2.44). В общем случае нужно стараться избегать слишком большого количества условий, чтобы не усложнять диаграмму. При частом использовании условных операторов (IF), вложенных в другие условные операторы, объем информации на диаграмме становится избыточным. С другой стороны, иногда бывает необходимо показать некоторую условную логику. Нужно стараться сохранять равновесие между двумя крайностями. Пока диаграмма остается легко читаемой и понимаемой, все в порядке. При усложнении условий создайте дополнительные диаграммы Последовательности: одну для описания событий, происходящих при выполнении условия IF, другую – для ELSE.

Помимо операторов IF, скрипты позволяют показывать на диаграмме

циклы и другие элементы псевдокода. Из них не генерируется код, но благодаря ним разработчики понимают логику развития событий на диаграмме.

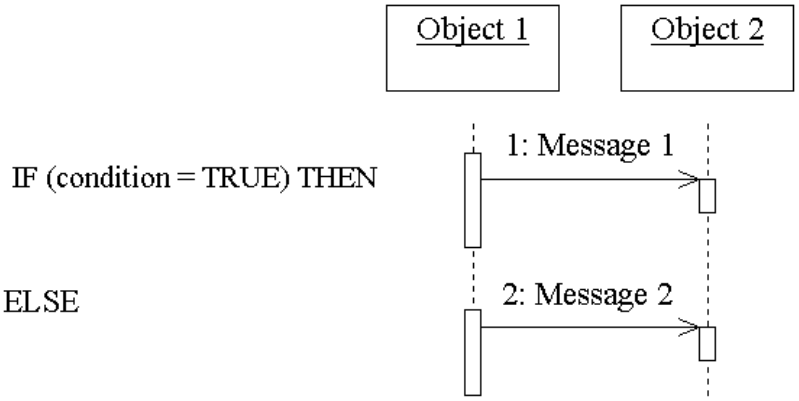


Рис. 2.44. Использование скриптов на диаграмме Последовательности

При разработке диаграмм Взаимодействия часто применяется ***двухэтапный подход***. Прежде всего, отображается информация высокого уровня, которая нужна конечным пользователям проектируемой системы. Сообщения еще не соотносятся с операциями, и объекты могут быть не соотнесены с классами. Эти диаграммы позволяют аналитикам, пользователям и всем, заинтересованным в бизнес-процессах лицам увидеть, как будут развиваться события в системе.

На втором этапе, после того как пользователи придут к согласию по поводу полученной диаграммы, можно углубиться в детали. При этом диаграмма может утратить свою полезность для пользователей, но станет, важна для разработчиков, тестировщиков и остальных участников команды проекта.

В начале второго этапа на диаграмму помещают некоторые новые объекты. Как правило, на каждой диаграмме Взаимодействия имеется управляющий объект, отвечающий за управление последовательностью событий сценария. Все диаграммы Взаимодействия для некоторого варианта использования могут иметь один и тот же управляющий объект, так что будет только один объект, контролирующий все потоки информации варианта использования. Управляющий объект не реализует никаких бизнес-процессов, он лишь посылает сообщения другим объектам. Управляющий объект отвечает за координацию действий других объектов и за делегирование ответственности. По этой причине такие объекты называют еще ***объектами-менеджерами*** (manager object).

Преимущество использования управляющего объекта заключается в отделении бизнес-логики от логики, определяющей последовательность событий. Если надо будет изменить последовательность, это затронет только

управляющий объект.

После добавления управляющего объекта диаграмма Последовательности, скорее всего, будет напоминать показанную на рис. 2.45.

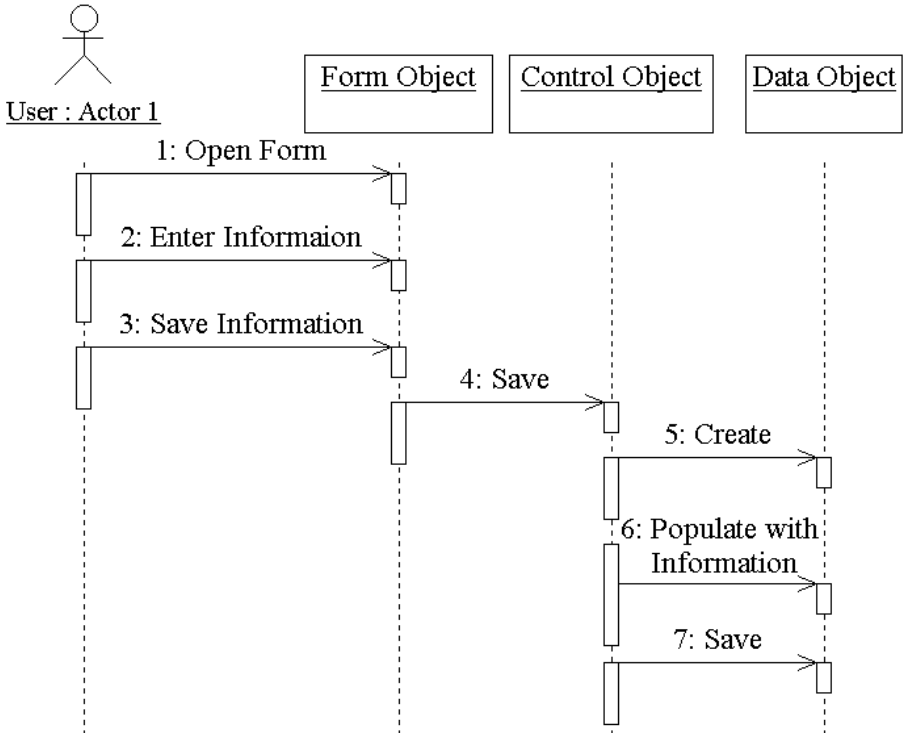


Рис. 2.45. Диаграмма Последовательности с управляющим объектом

Можно поместить на диаграмму и другие объекты, отвечающие за безопасность, обработку ошибок или за связь с базой данных. Многие из них бывают настолько общими, что допускают повторное использование во многих приложениях. Рассмотрим для примера работу с базой данных.

При сохранении информации в базе данных или при получении ее оттуда чаще всего используются две возможности. Допустим, мы сохраняем в базе информацию о новом сотруднике Джо Доу. Объект Джо Доу может знать о базе данных, в этом случае он сохраняет себя сам. Но он может быть полностью отделен от логики базы данных, и тогда его сохранением должен заниматься другой объект. Начнем со случая, когда Джо знает о базе данных. В этом примере логика приложения не отделена от логики базы данных (рис.2.46). Объект Джо Доу занимается как первым (прием на работу или увольнение Джо Доу), так и вторым (например, сохранение информации о Джо в базе данных и получение нее оттуда при необходимости). Поскольку множество объектов в таком приложении содержит логику базы данных, последствия любого ее изменения скажутся практически на всех этих объектах,

это затронет все приложение. С другой стороны, такой подход легче моделировать и реализовывать.

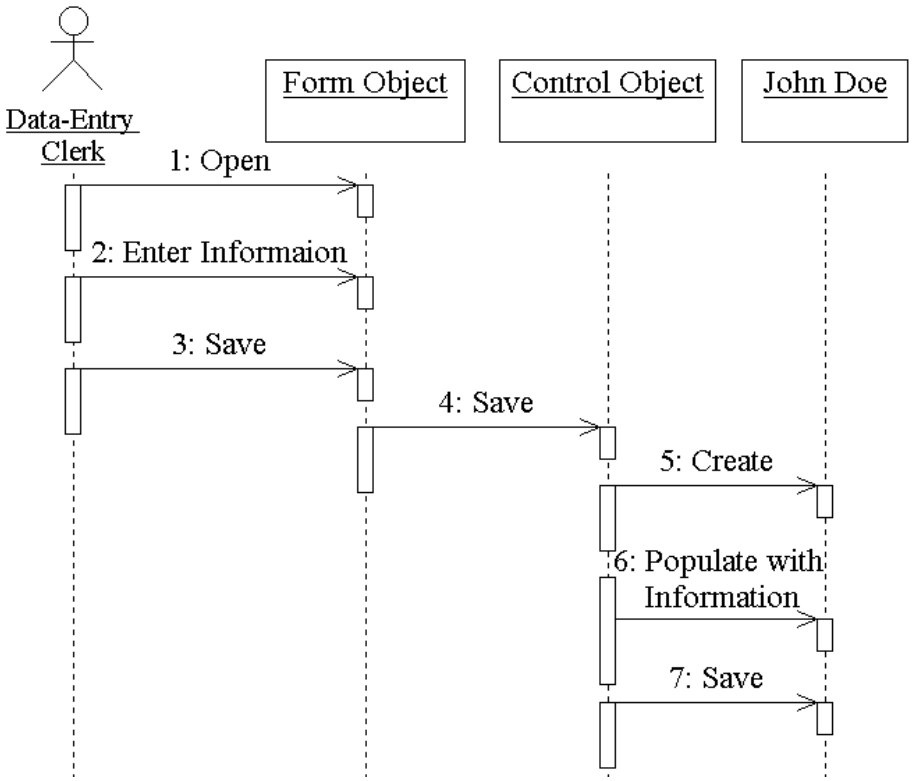


Рис. 2.46. Интеграция логики приложения и логики базы данных

Другая возможность заключается в отделении логики приложения от логики базы данных. В этом случае для работы с базой нужно создать объект, который можно назвать ***менеджером транзакций*** (transaction manager). Объект Джо Доу по-прежнему содержит бизнес-логику. Он знает, как принять на работу, уволить Джо или как повысить его по службе. Менеджер транзакций знает, как затребовать информацию о Джо из базы данных или как сохранить ее там. Соответствующая диаграмма Последовательности может иметь вид, представленный на рис. 2.47.

Преимуществом данного подхода является возможность повторного использования объекта Джо Доу в другом приложении с другой базой данных или вообще без базы данных. Кроме того, уменьшаются последствия изменений в требованиях к системе. Изменения в базе данных не повлияют на логику приложения, и наоборот. Недостаток заключается в больших затратах времени на моделирование и реализацию этого решения.

Два описанных подхода являются наиболее общими при работе с базой данных, хотя существуют и другие приемы, облегчающие этот процесс. Ка-

кое бы решение Вы ни приняли, следите за тем, чтобы процесс нашел свое отражение на диаграммах Последовательности.

Помимо работы с базами данных, можно добавить в систему и другие объекты, отвечающие за обработку ошибок, проблемы безопасности или коммуникацию между процессами. Все эти детали не интересуют клиента, но неоценимы для разработчиков.

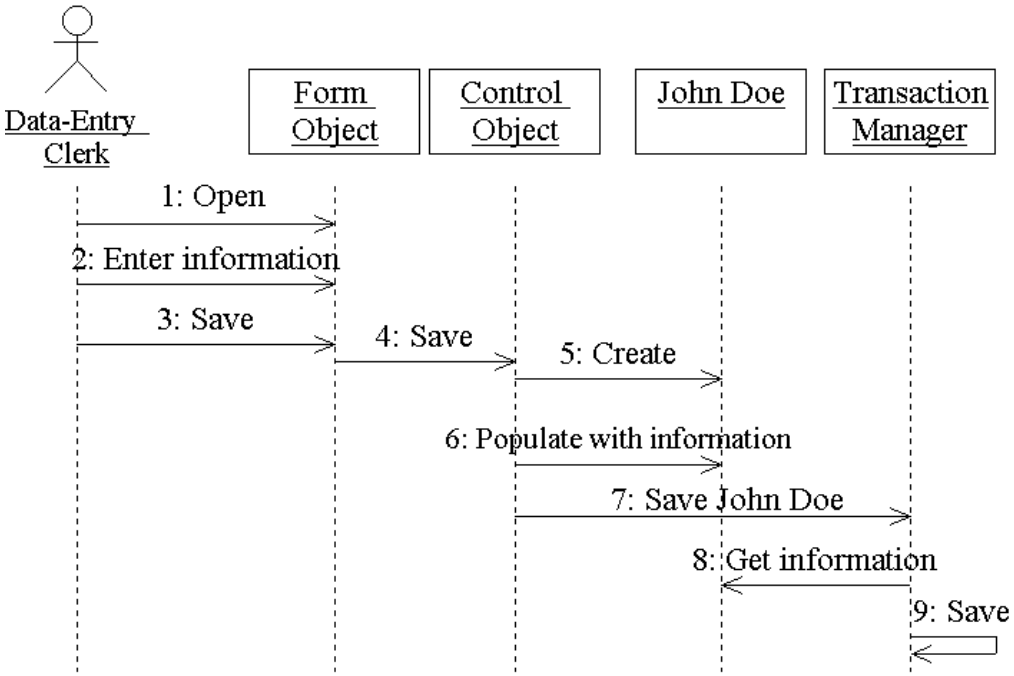


Рис. 2.47. Разделение логики приложения и логики базы данных

В качестве примера рассмотрим построение диаграмм Взаимодействия для системы обработки заказов. Высший приоритет среди пользователей имеет вариант использования «Ввести новый заказ», он же связан с наибольшим риском. Создадим основной поток событий для варианта использования «Ввести новый заказ». Он будет иметь следующий вид.

1. *Продавец выбирает в имеющемся меню пункт «Создать новый заказ».*
2. *Система выводит форму «Детали заказа».*
3. *Продавец вводит номер заказа, заказчика и то, что заказано.*
4. *Продавец сохраняет заказ.*
5. *Система создает новый заказ и сохраняет его в базе данных.*

Составим описание сценария для этого варианта использования.

1. *Продавец вводит новый заказ.*
2. *Продавец пытается ввести заказ, но товара нет на складе.*

3. Продавец пытается ввести заказ, но при его сохранении в базе данных возникает ошибка.

Для составленного сценария создадим диаграмму Последовательности и Кооперативную диаграмму. Будем рассматривать успешный ход событий, когда продавец правильно вводит новый заказ.

1. Добавим на диаграмму основные действующие лица и объекты. Действующим лицом на данной диаграмме будет Salesperson (Продавец), а основными объектами - Order Options Form (Выбор варианта заказа), Order Detail Form (Форма деталей заказа), Order N1234 (Заказ №1234).

2. Добавим на диаграммы следующие сообщения:

1) Create new order (Создать новый заказ);
2) Open form (Открыть форму) - между Order Options Form и Order Detail Form;
3) Enter order number, customer, order items (Ввести номер заказа, заказчика и число заказываемых предметов) - между Salesperson и Order Detail Form;
4) Save the order (Сохранить заказ) - между Salesperson и Order Detail Form;
5) Create new, blank order (Создать пустой заказ) - между Order Detail Form и Order N1234;
6) Set the order number, customer, order items (Ввести номер заказа, заказчика и число заказываемых предметов) - между Order Detail Form и Order N1234;
7) Save the order (Сохранить заказ) - между Order Detail Form и Order N1234.

4. Добавим на диаграмму дополнительные объекты. Как видно из составленных диаграмм, объект Order Detail Form имеет множество ответственностей, с которыми лучше всего мог бы справиться управляющий объект. Кроме того, новый заказ должен сохранять себя в базе данных сам. Вероятно, эту обязанность лучше было бы переложить на другой объект.

Поместим между объектами Order Detail Form и Order N1234 новый объект - Order Manager (Управляющий заказами). А справа от Order N1234 расположим объект Transaction Manager (Управляющий транзакциями).

5. Назначим ответственности объектам. Поместим на диаграмму новое сообщение Save the order (Сохранить заказ), расположив его под одноименным сообщением 4 между Order Detail Form и Order Manager.

Добавим сообщения с шестого по девятое, назвав их:

1) Create new, blank order (Создать новый заказ) - между Order Manager и Order N1234;
2) Set the order number, customer, order items (Вести номер заказа, заказ-

чика и число заказываемых предметов) - между Order Manager и Order N1234;

3) Save the order (Сохранить заказ) - между Order Manager и Transaction Manager;

4) Collect order information (Информация о заказе) - между Transaction Manager и Order N1234.

На линии жизни объекта Transaction Manager (Управляющий транзакциями) ниже сообщения 9, добавим рефлексивное сообщение Save the order information to the database (Сохранить информацию о заказе в базе данных).

6. Произведем соотнесение объектов с классами. Класс Order Options Form (Выбор варианта заказа) соотнесем с объектом OrderOptions (Выбор заказа). Класс OrderDetail с объектом Order Detail Form. Класс OrderMgr - с объектом Order Manager. Класс Order - с объектом Order N1234. Класс Transaction Mgr- с объектом Transaction Manager.

Полученные диаграмма Последовательности и Кооперативная диаграмма представлены на рис. 2.48 и рис. 2.49 соответственно.

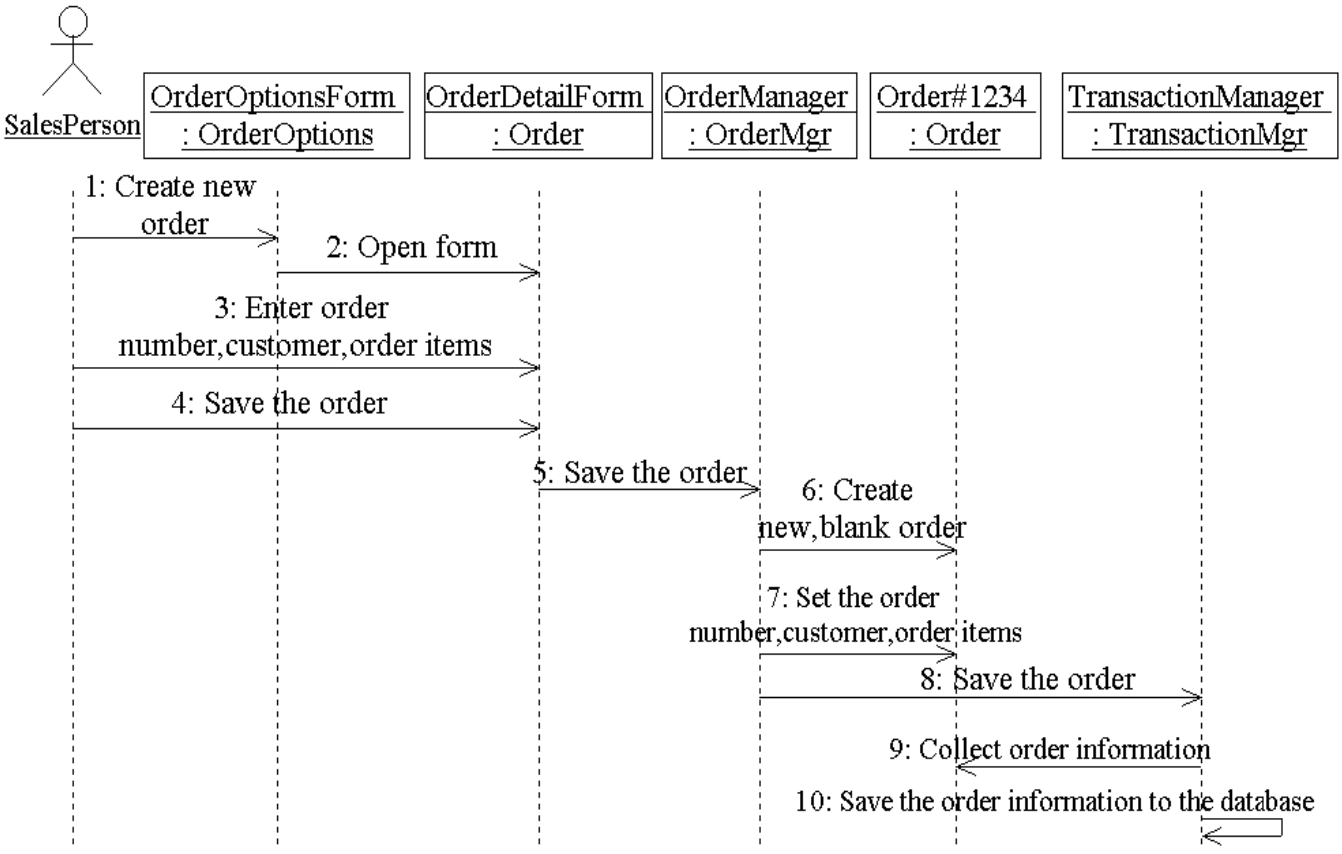


Рис. 2.48. Диаграмма Последовательности с именами классов

7. Произведем соотнесение сообщений с операциями. Сообщении 1: Create new order (Создать новый заказ) соотнесем с операцией Create(). Сообщение 2: Open form (Открыть форму) - с операцией Open(). Сообщение 3: Enter order number, customer, order items (Ввести номер заказа, заказчика и число заказываемых предметов) - с операцией Submitlnfo(). Сообщение 4: Save the order (Сохранить заказ) - с операцией Save(). Сообщение 5: Save the order (Сохранить заказ) - с операцией Save0rder().Сообщение 6: Create new, blank order (Создать пустой заказ) - с операцией CreateO. Сообщение 7: Set the order number, customer, order items (Ввести номер заказа, заказчика и число заказываемых предметов) - с операцией SetlnfoQ. Сообщение 8: Save the order (Сохранить заказ) - с операцией Save0rder(). Сообщение 9: Collect order information (Информация о заказе) - с операцией GetlnfoQ. Сообщение 10: Save the order information to the database (Сохранить информацию о заказе в базе данных) - с операцией Commit().

После проделанных действий диаграммы должна выглядеть, как показано на рис. 2.50 и рис. 2.51.

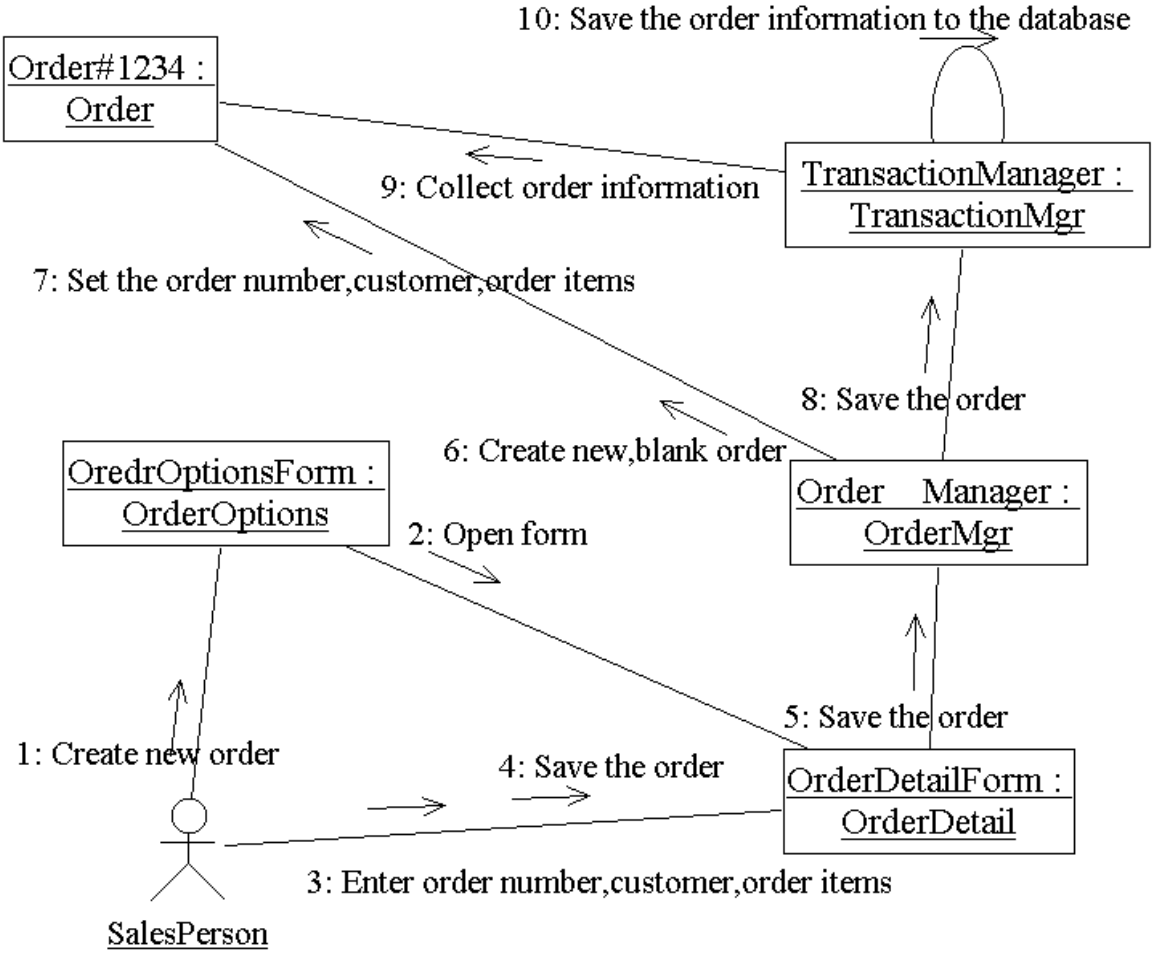


Рис. 2.49. Кооперативная диаграмма с именами классов

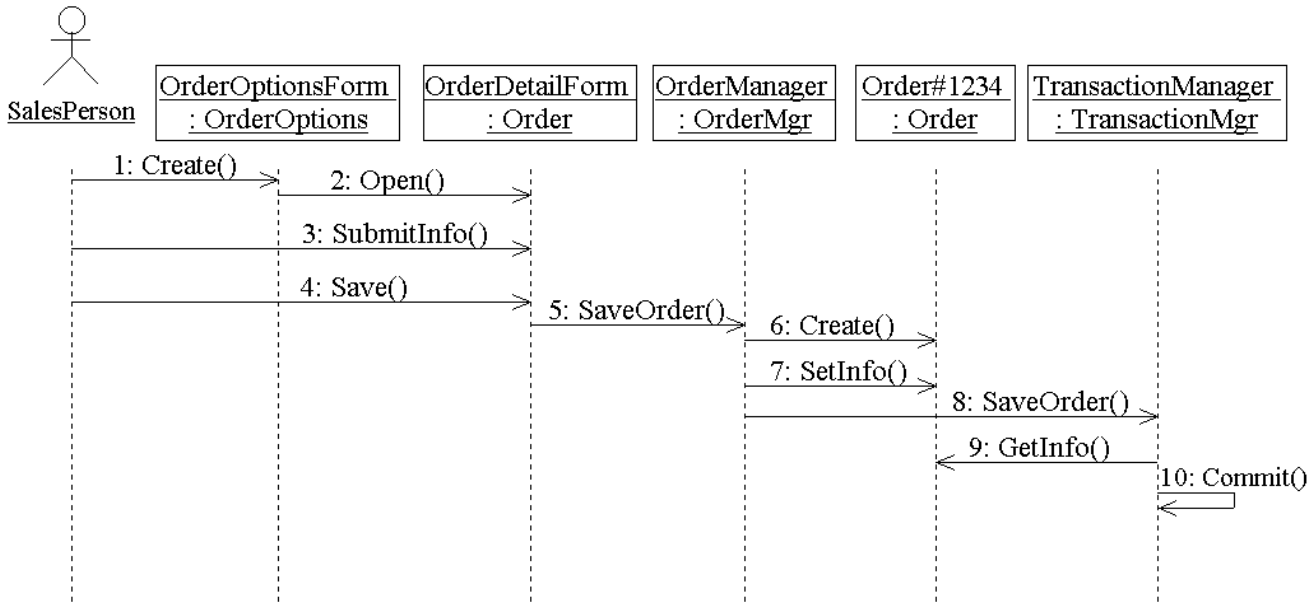


Рис. 2.50. Диаграмма Последовательности с показанными на ней операциями

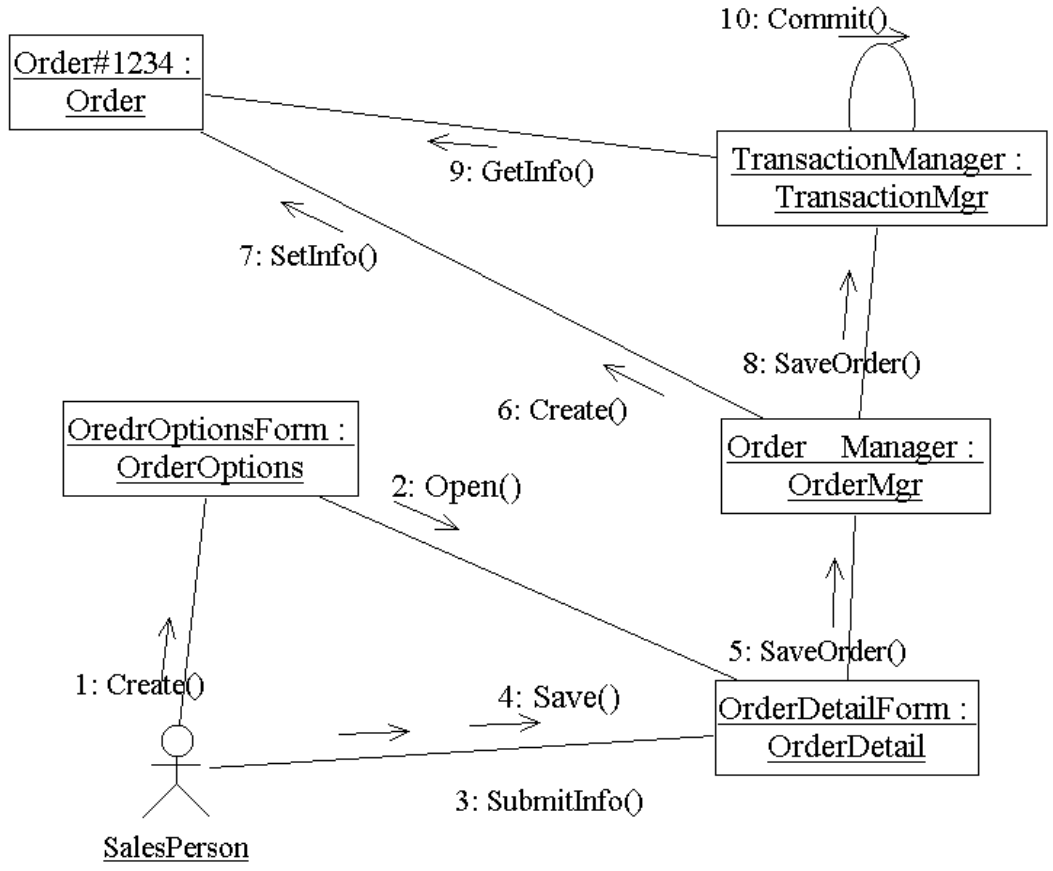


Рис. 2.51. Кооперативная диаграмма с показанными на ней операциями

2.7.5. Диаграммы Классов

Диаграммы Классов позволяют показать основные характеристики классов системы и связи между классами. Диаграммы Классов можно создавать для отдельного варианта использования, всей системы или подсистемы. С помощью этих диаграмм аналитики могут показать детали системы. Если какой-либо класс несет слишком большую функциональную нагрузку, это будет видно на диаграмме классов, и архитектор сможет перераспределить ее между другими классами. С помощью такой диаграммы можно также выявить случаи, когда между сообщающимися классами не определено никаких связей. Case-средства такие как Rational Rose на основе диаграмм классов генерируют основу кода классов, которую программисты затем дополняют деталями на выбранном ими языке.

Основные вопросы, связанные с созданием диаграмм классов, рассматривались в разделе «Классы». В данном разделе, используя приведенные сведения, рассмотрим пример создания диаграммы классов для системы обработки заказов. Для выявления классов воспользуемся созданными диаграммами Взаимодействия.

1. Объединим обнаруженные классы в пакеты по стереотипу. Для этого создадим пакеты Entities (Сущности), Boundaries (Границы) и Control (Управление). Отобразим результаты на Главной диаграмме Классов (рис. 2.52.).

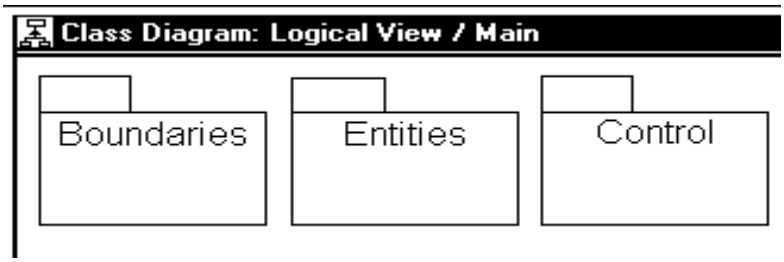


Рис. 2.52. Главная диаграмма Классов системы обработки заказов

2. Создадим диаграмму Классов варианта использования «Ввести новый заказ». Для этого поместим на диаграмму классы, представленные на диаграммах Взаимодействия: OrderOptions, OrderDetail, Order, OrderMgr, TransactionMgr. Полученная диаграмма Классов приведена на рис. 2.53.

3. Добавим к классам следующие стереотипы:

1) классам OrderOptions и OrderDetail добавим стереотип Boundary;
2) классам OrderMgr и TransactionMgr - стереотип Control;
3) классу Order - стереотип Entity.

Теперь диаграмма Классов будет иметь вид, приведенный на рис. 2.54.

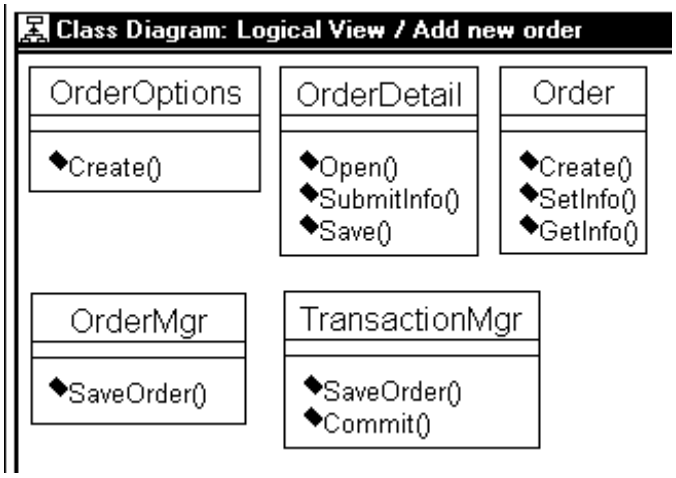


Рис. 2.53. Диаграмма Классов «Ввести новый заказ»

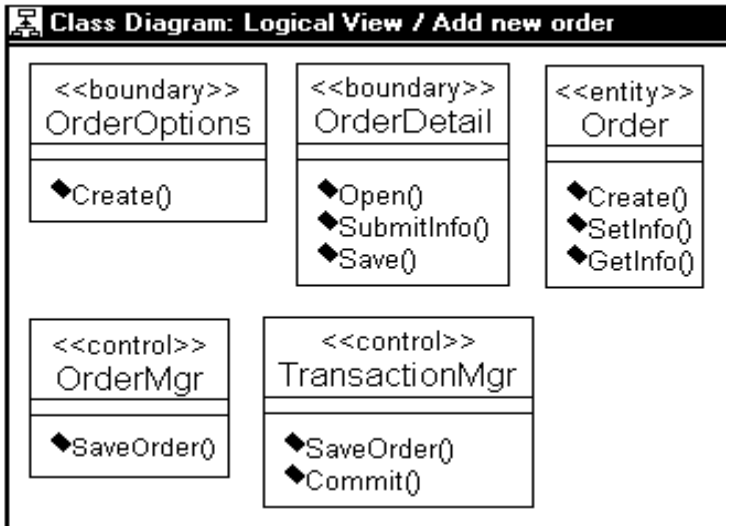


Рис. 2.54. Стереотипы классов варианта использования «Ввести новый заказ»

4. Распределим классы по пакетам в соответствии с их стереотипами.

В пакет Boundaries попадут классы OrderOptions и OrderDetail. В пакет Control - классы OrderMgr и TransactionMgr. В пакет Entities класс Order.

5. Для определения атрибутов необходимо вновь обратится к потоку событий. В результате к классу Order диаграммы Классов добавим атрибуты Order Number (Номер заказа) и Customer Name (Имя клиента). Просмотрим список заказываемых товаров. Так как в одном заказе можно указать большое количество товаров и у каждого из них имеются свои собственные данные и поведение, можно смоделировать товары как самостоятельные классы, а не как атрибуты класса Order.

Чтобы привести модель в соответствие с новыми идеями обновим диаграмму Последовательности (рис. 2.55).

6. Предположим, что в этот момент заказчик решил изменить требования: «Нам надо отслеживать дату заказа и дату его выполнения. Кроме того, так как у нас появились новые поставщики, слегка изменилась процедура инвентаризации».

Изменения требований сильно повлияют на вариант использования «Провести инвентаризацию» (работа с которым пока отложена). В варианте использования «Ввести новый заказ» новые требования, связанные с датами, приведут к тому, что в класс Order добавятся два атрибута. Модель вновь стала соответствовать последним требованиям.

7. Добавим новый класс OrderItem. Назначим этому классу стереотип Entity.

8. Добавим на диаграмму Классов «Ввод нового заказа» атрибуты и покажем их сигнатуры.

В классе Order введем новые атрибуты:

1) OrderNumber: Integer;
2) CustomerName: String;.
3) OrderDate: Date;
4) OrderFillDate: Date.

В классе OrderItem введем новые атрибуты:

1) ItemID: Integer;
2) ItemDescription: String.

9. Добавим к классу OrderItem следующие операции: Create, SetInfo, Getlnfo.

10. Выбрав язык реализации можно добавить к классам параметры операций, типы данных и типы возвращаемых значений (т.е. сигнатуры операций). Введем следующие сигнатуры операций.

Для класса Order:

1) Create( ): Boolean;
2) SetInfo(OrderNum: Integer, Customer: String, OrderDate: Date, FillDate: Date): Boolean;
3) GetInfo( ): String.

Для класса OrderItem:

1) Getlnfo( ): String;
2) SetInfo(ID: Integer = 0): Boolean;
3) Create( ): Boolean.

Для класса OrderDetail:

1) Open( ): Boolean;
2) SubmitInfo( ): Boolean;
3) Save( ): Boolean.

Для класса OrderOptions: Create( ): Boolean.

Для класса OrderMgr:

SaveOrder(OrderID: Integer): Boolean.

Для класса TransactionMgr:

1) SaveOrder(OrderID: Integer: Boolean;
2) Commit( ): Integer.

11. Определим и добавим связи к классам, принимающим участие в варианте использования «Ввести новый заказ». Для этого просмотрим диаграммы Последовательности. Ограничимся добавлением ассоциаций.

Установим однонаправленную ассоциацию между классами:

1) OrderOptions и OrderDetail;
2) OrderDetail и OrderMgr;
3) OrderMgr и Order;
4) OrderMgr и TransactionMgr;
5) TransactionMgr и Order;
6) TransactionMgr и Orderltem;
7) Order и Orderltem.

Между классами OrderOptions и OrderDetail со стороны класса OrderOptions установим множественность Multiplicity > Zero or One (Множественность > Нуль или один). На другом конце однонаправленной ассоциации - Multiplicity > Zero or One (Множественность > Нуль или один). Аналогично добавим на диаграмму значения множественности для остальных ассоциаций (рис. 2.56).

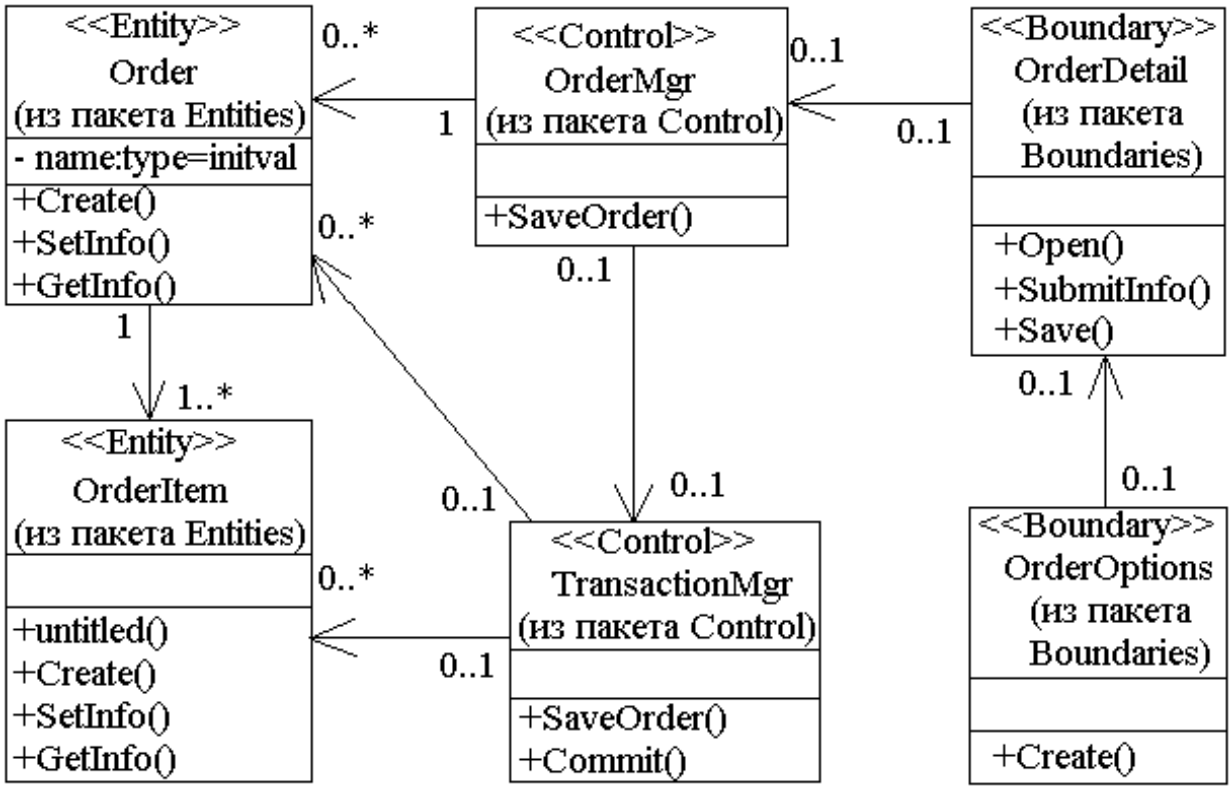


Рис. 2. 56. Ассоциация сценария «Ввести новый заказ»

2.7.6. Диаграммы Состояний

Диаграмма Состояний (State Transition) содержит информацию о состояниях, в которых может находиться объект, о том, как он переходит из одного состояния в другое и каким образом он ведет себя в этих состояниях. На диаграмме Состояний отображают жизненный цикл одного объекта, начиная с момента его создания и заканчивая разрушением. С помощью таких диаграмм удобно моделировать динамику поведения класса. Диаграммы Состояний не нужно создавать для каждого класса, они применяются только в очень сложных случаях. Если объект класса может существовать в нескольких состояниях и в каждом из них ведет себя по-разному, тогда для него требуется такая диаграмма. Однако во многих проектах они вообще не используются.

Состоянием (state) называется одно из возможных условий, в которых может существовать объект. Находясь в конкретном состоянии, объект может выполнять определенные действия. Например, он может генерировать отчет, осуществлять некоторые вычисления или посылать сообщение другому объекту. Для выявления состояний объекта необходимо исследовать две области модели: значения атрибутов объекта и связи с другими объектами.

В нотации UML состояние изображают в виде прямоугольника с закругленными краями. На рис. 2.57 можно видеть следующие состояния: Открыт, Превышен счет, Закрыт.

С состоянием можно связать данные пяти типов: деятельность, входное действие, выходное действие, событие и история состояния. Рассмотрим каждый из них в контексте примера. На рис. 2.57 показана диаграмма Состояний для класса Account (Счет) системы ATM.

Деятельностью (activity) называется поведение, реализуемое объектом, когда он находится в определенном состоянии. Деятельность - это прерываемое поведение. Оно может выполняться до своего завершения, если объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должны предшествовать слово do (делать) и двоеточие. На рис. 2.77 показано, что у состояния «Превышен счет» имеется деятельность «Послать уведомление клиенту».

Входным действием (entry action) называется поведение, которое выполняется, когда объект переходит в определенное состояние. Оно осуществляется не после того, как объект перешел в состояние, а является частью перехода. В отличие от деятельности, данное действие рассматривается как непрерываемое. Входное действие показывают внутри состояния, ему предшествуют слово entry (вход) и двоеточие. Например, при переходе объекта Счет в состояние «Превышен счет» (рис.2.57), выполняется действие «Вре-

менно заморозить счет» независимо от того, откуда объект переходит в это состояние.

Выходное действие (exit action) осуществляется как составная часть процесса выхода из состояния. Оно является частью процесса перехода. Как и входное, выходное действие является непрерываемым. Выходное действие изображают внутри состояния, ему предшествуют слово exit (выход) и двоеточие. При выходе объекта Счет из состояния «Превышен счет» (рис 2.57), независимо от того, куда он переходит, выполняется действие «Разморозить счет».

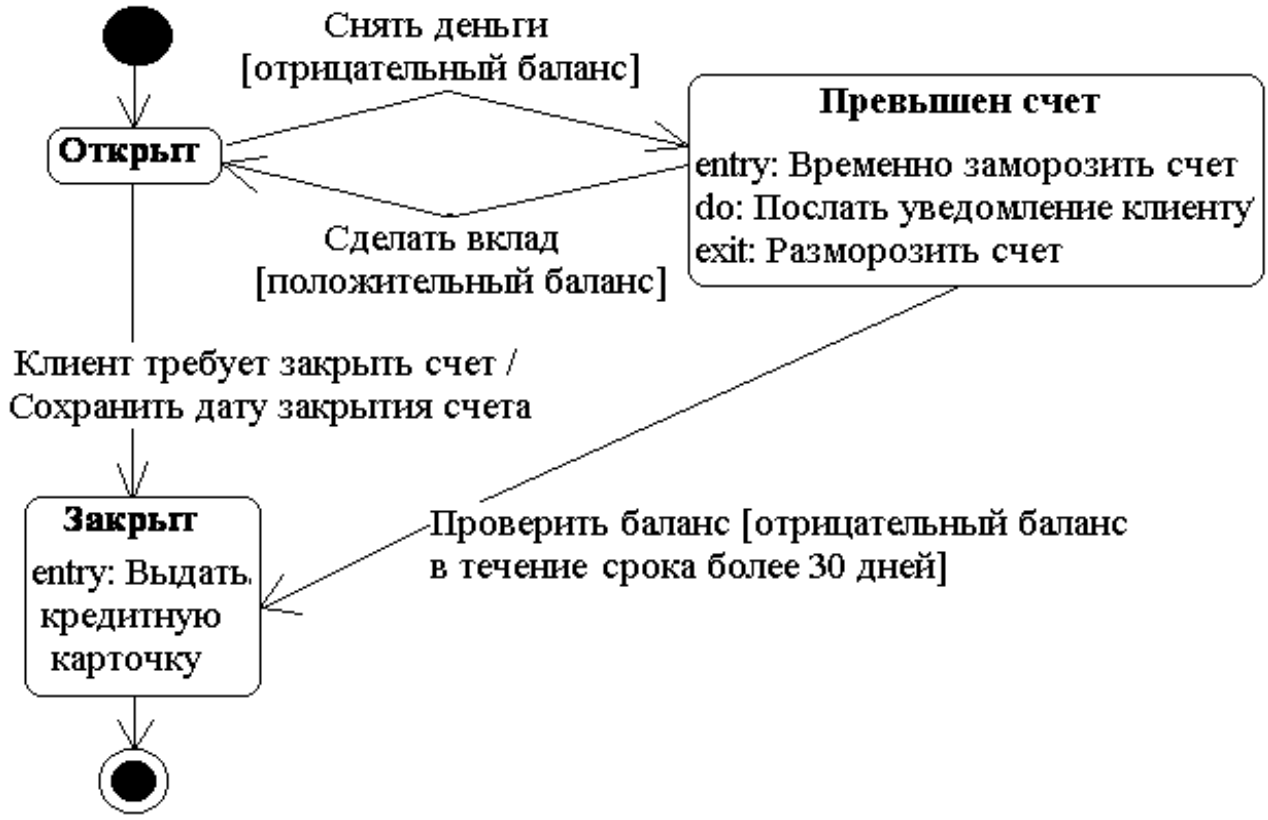


Рис. 2.57. Диаграмма Состояний для класса Account (Счет) системы ATM

Переходом (transition) называется перемещение объекта из одного состояния в другое. На диаграмме переходы изображают в виде стрелки, начинающейся в первоначальном и заканчивающейся в последующем состоянии. Переходы могут быть рефлексивными: объект переходит 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: события, аргументы, ограждающие условия, действия и посылаемые события. Рассмотрим эти

параметры в контексте примера АТМ (рис. 2.57).

Событие (event) - это то, что вызывает переход из одного состояния в другое. В нашем примере событие «Клиент требует закрыть счет» вызывает переход счета из открытого состояния в закрытое. Событие размещают на диаграмме вдоль линии перехода. Для отображения события на диаграмме можно использовать как имя операции, так и обычную фразу. Если использовать операции, то событие «Клиент требует закрыть счет» 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 должно иметь события, так как именно они инициируют переход. Тем не менее, бывают и ***автоматические переходы***, не имеющие событий. При этом объект сам перемещается из одного состояния в другое со скоростью, предусматривающей выполнение входных действий деятельности и выходных действий. В этом случае на диаграмме вдоль линии перехода нет никаких событий.

Ограждающее условие (guard conditions) определяет, когда переход может быть выполнен, а когда нет. На диаграмме ограждающие условия заключают в квадратные скобки и размещают, вдоль линии перехода после имени события. В нашем примере событие «Сделать вклад» переведет счет из состояния «Превышение счета» в состояние «Открыт», только если баланс больше нуля. В противном случае переход не осуществится. 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 выбран автоматически.

Действием (action) является непрерываемое поведение, выполняющееся как часть перехода. Входные и выходные действия показывают внутри состояния, поскольку они определяют, что происходит, когда объект входит или выходит из состояния. Другие действия изображают вдоль линии перехода, так как они не должны осуществляться при входе или выходе из состояния. Действие размещают вдоль линии перехода после имени события, ему предшествует косая черта (/). 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».

Поведение объекта во время деятельности, входных и выходных действий может включать в себя отправку события другому объекту. Например, объект Account (Счет) может посылать событие объекту Card reader (Устрой-

ство чтения карты). В этом случае описанию деятельности, входного или выходного действия предшествует знак «^» (рис. 2.58). Здесь цель - это объект, получающий событие, событие - посылаемое сообщение, а аргументы являются параметрами посылаемого сообщения. При получении объектом события будет выполнена определенная деятельность.

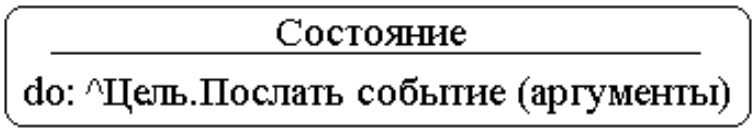


Рис. 2.58. Событие, посылаемое другому объекту

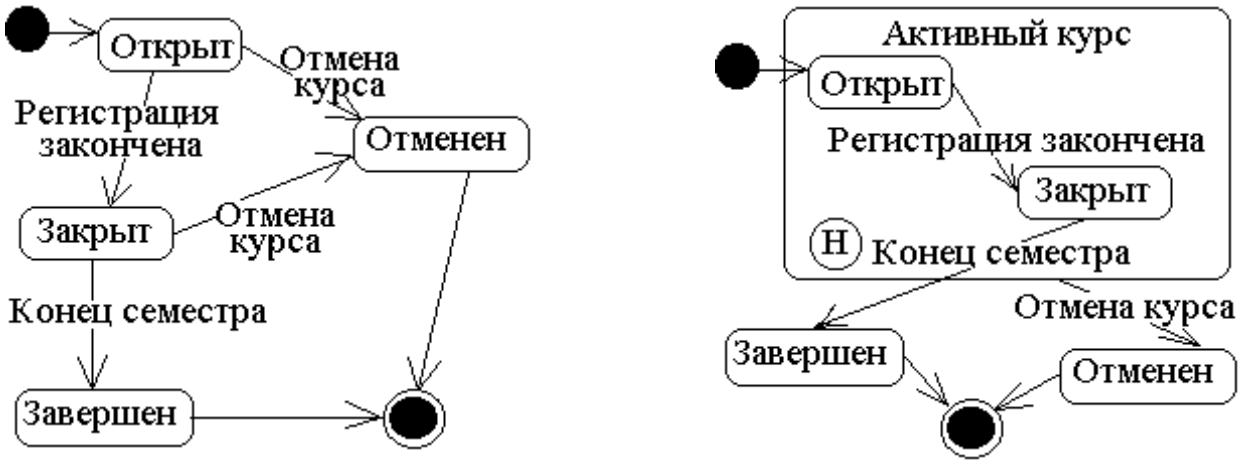
На диаграмму добавляют два специальных состояния объекта: начальное и конечное. ***Начальным*** (start) называется состояние, в котором объект находится сразу после своего создания. Начальное состояние обязательно - читатель должен знать, с чего начинается объект. На диаграмме может быть только одно начальное состояние, его изображают в виде закрашенного кружка. На рис. 2.57 оно показано до состояния «Открыт». ***Конечным*** (stop) называется состояние, в котором объект находится непосредственно перед уничтожением. Конечные состояния не являются обязательными, их может быть сколько угодно. Конечное состояние изображают в виде закрашенного кружка с ободком («бычий глаз»). На рис. 2.57 оно показано после состояния «Закрыт».

Для уменьшения беспорядка на диаграмме можно вкладывать состояния одно в другое. Вложенные состояния называются ***подсостояниями*** (substates), а те, в которые они вложены, - ***суперсостояниями*** (superstates).

Если у нескольких состояний имеются идентичные переходы, эти состояния можно сгруппировать вместе в суперсостояние. Затем, вместо того чтобы поддерживать одинаковые переходы (по одному на каждое состояние), их можно объединить, перенеся в суперсостояние. На рис. 2.59, а приведен пример диаграммы без вложенных состояний. На рис. 2.59, б изображена та же диаграмма с использованием вложенных состояний. Суперсостояния позволяют «навести порядок» на диаграмме Состояний.

Бывают случаи, когда система должна помнить, в каких состояниях она была в прошлом. Если, например, выход из суперсостояния с тремя подсостояниями, может потребовать, чтобы система запомнила, из какой точки суперсостояния произошел выход. Существует два способа решения этой проблемы. Во-первых, можно включить в суперсостояние начальное состояние. В таком случае будет понятно, где находится стартовая точка в суперсостоянии. Именно там окажется объект при входе в суперсостояние. Во-вторых, чтобы

запомнить, где был объект, можно использовать историю состояний (state history). В таком случае объект может выйти из суперсостояния, а затем вернуться точно в то место, откуда вышел. При подключении истории состояний в углу диаграммы располагается буква «H» в кружке (рис.2.60).



а) б)

Рис. 2.59. Диаграмма Состояний:
а) без вложенных состояний; б) суперсостояния позволили «внести порядок»

Рис. 2.60. История суперсостояния

2.7.7. Диаграммы Компонентов

Диаграммы Компонентов показывают, как выглядит модель на физическом уровне. На ней изображаются компоненты программного обеспечения системы и связи между ними.

Компонентом (component) называется физический модуль кода. Существуют два основных типа компонентов: библиотеки исходного кода и исполняемые компоненты. Например, для языка C++ файлы с расширением «cpp» и «h» будут отдельными компонентами. Получающийся при компиляции исполняемый файл с расширением «exe» также является компонентом системы.

Компонент (Component) соответствует программному модулю с хорошо определенным интерфейсом. Ниже обсуждаются различные стереотипы компонентов.

Спецификация и тело подпрограммы (Subprogram Specification and Body) представляют видимую спецификацию подпрограммы и тело ее реализации (рис. 2.61, а, б). Обычно подпрограмма состоит из стандартных программных компонентов (subroutines) и не содержит определений класса.

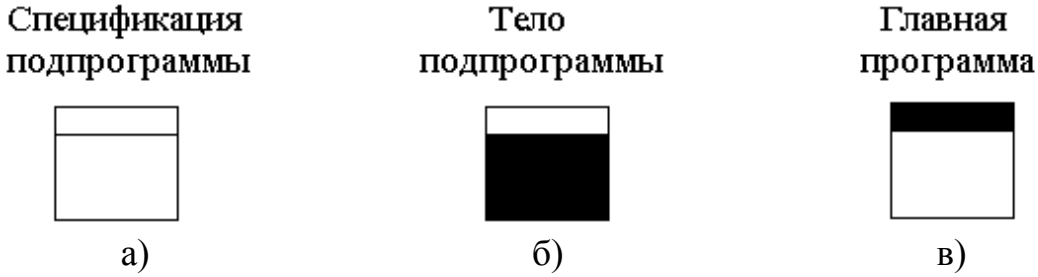


Рис. 2.61. Графическая нотация для изображения:
а) спецификации подпрограммы; б) тела подпрограммы; в) главной программы

Главная программа (Main Program) - это файл, содержащий корень программы (рис. 2.61, в). Например, в среде PowerBuilder такой файл содержит объект приложения.

Спецификация и тело пакета (Package Specification and Body) изображаются с помощью нотации показанной на рис. 2.62, а и б. Пакет в данном случае - это реализация класса. Спецификацией пакета является заголовочный файл со сведениями о прототипах функций для класса. На C++ это файл с расширением «h». Тело пакета содержит код операций класса. На C++ это файл с расширением «cpp».

Исполняемые компоненты - это исполняемые файлы, файлы DLL и задачи.

Файл динамической библиотеки (файл DLL) изображается с помощью нотации показанной на рис. 2.62, в.

Спецификация и тело задачи (Task Specification and Body) отображают пакеты, имеющие независимые потоки управления (рис. 2.62, г, д). Испол-

няемый файл обычно представляют как спецификацию задачи с расширением «exe».

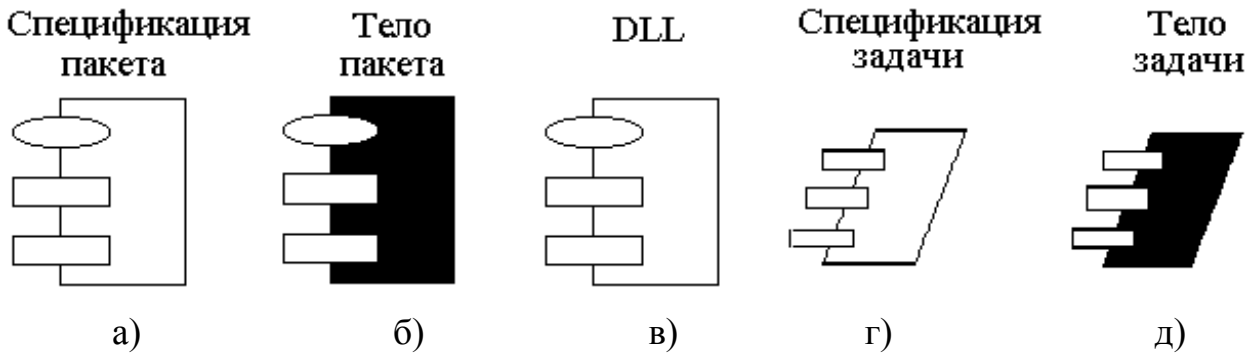


Рис.2. 62. Графическая нотация для изображения:
а) спецификации пакета; б) тела пакета; в) DLL; г) спецификации задачи; д) тела задачи

Единственный возможный тип связей между компонентами - это зависимость. Он показывает, что один из компонентов должен компилироваться перед началом компиляции другого. Зависимость между компонентами изображают пунктирной линией.

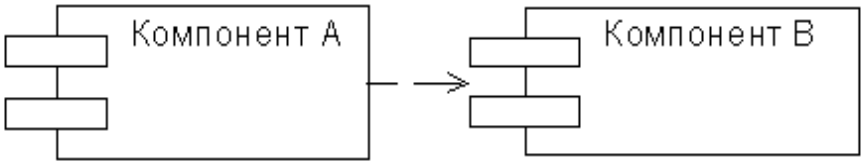


Рис. 2.63. Графическое представление связи между компонентами

На рис. 2.63 показан пример зависимости. В этом случае компонент А зависит от В. Это может означать, что в компоненте А существует некоторый класс, зависящий от какого-то класса компонента В. Знание о зависимостях важны при компиляции. Так как А зависит от В, А не может быть скомпилирован до В. Анализируя эту диаграмму, можно понять, что сначала компилируется В, а затем уже А. Следует избегать циклических зависимостей между компонентами. Если А зависит от В, а В от А, то ни один из них нельзя компилировать, пока не скомпилирован другой. Таким образом, оба компонента должны рассматриваться как один большой компонент. Все циклические зависимости необходимо устранить до начала генерации кода.

Зависимости связаны также с проблемами управления системой. Если А зависит от В, то любые изменения в В повлияют на А. С помощью диаграммы Компонентов персонал сопровождения системы может оценить последствия

вносимых изменений. Если компонент зависит от большего числа других компонентов, велика вероятность того, что его затронут изменения в системе.

Наконец, зависимости дают возможность понять, какие части системы можно использовать повторно, а какие нельзя. В нашем примере А трудно будет применить второй раз. Поскольку он зависит от В, то сделать это можно только совместно с В. С другой стороны, В легко использовать повторно, так как он ни от чего не зависит. Чем от меньшего числа компонентов зависит данный, тем легче его будет использовать повторно.

Перед началом генерации кода необходимо соотнести каждый из файлов с соответствующими компонентами.

Диаграммы Компонентов применяются теми участниками проекта, которые отвечает за компиляцию системы. Они нужна там, где начинается генерация кода. Диаграмма Компонентов дает представление о том, в каком порядке нужно компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. Эта диаграмма показывает соответствие классов реализованным компонентам.

На рисунках 2.64 и 2.65 представлены примеры диаграмм Компонентов для клиента и сервера ATM.

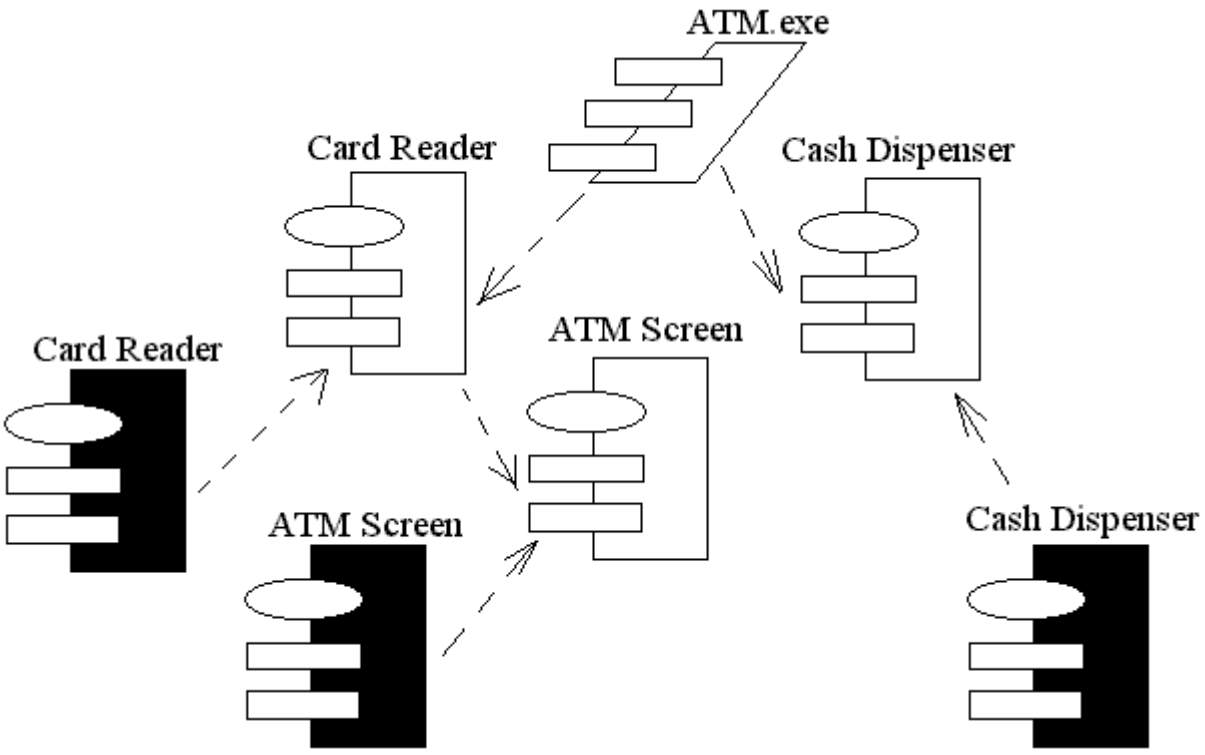


Рис. 2.64. Диаграмма Компонентов для клиента ATM

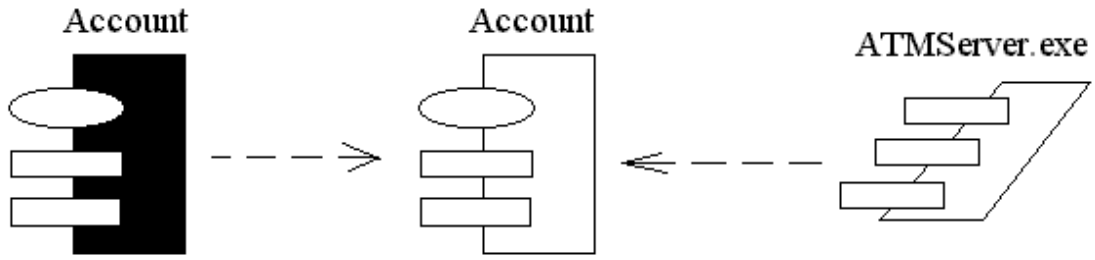


Рис. 2.65. Диаграмма Компонентов для сервера ATM

2.7.8. Диаграммы Размещения

На ***диаграмме Размещения*** (Deployment diagram) показывают физическое расположение различных компонентов системы в сети. Она содержит процессоры, устройства, процессы и связи между процессорами и устройствами. Для системы может быть создана только одна диаграмма Размещения. Диаграмма Размещения нужна всем участникам проекта. Эксплуатационный персонал, например, исходя из нее, планирует работу по установке системы.

Процессором (processor) называется любая машина, имеющая вычислительную мощность, т.е. способная производить обработку данных. В эту категорию попадают серверы, рабочие станции и другие устройства, содержащие физические процессоры. Процессор изображается с помощью нотации показанной на рис. 2.66.

Рис. 2.66. Графическое нотация для изображения процессора и устройства

Устройством (device) называется аппаратура, не обладающая вычислительной мощностью. Это, например, принтеры, терминалы ввода/вывода (dumb terminals), сканеры т.п. Устройство изображается с помощью нотации показанной на рис. 2.66.

Процессоры и устройства называются также узлами (nodes) сети.

Связью (connection) называется физическая связь между двумя процессорами, двумя устройствами или процессором и устройством. Чаще всего

связи отражают физическую сеть соединений между узлами сети. Кроме того, это может быть ссылка Интернета, связывающая два узла.

Процессом (process) называется поток обработки информации (execution), выполняющийся на процессоре. Процессом, например, считается исполняемый файл. Процессы отображаются непосредственно под процессором (процессорами), на котором выполняются. На рис. 2.67 приведена Диаграмма Размещения для системы ATM

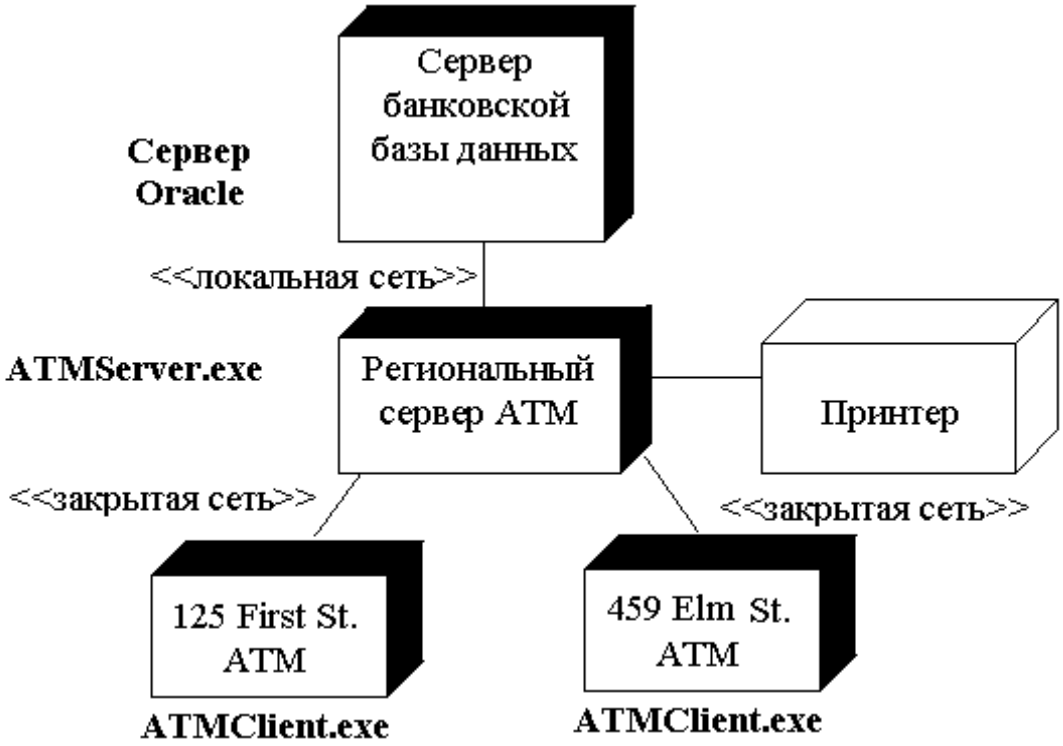


Рис. 2.67. Диаграмма Размещения для системы ATM

2.7.9. Примечания и пакеты

На диаграммах UML можно размещать примечания, содержащие дополнительную информацию об элементах диаграммы. Хотя примечания не влияют на генерируемый код, они помогают разработчикам и другим участникам проекта лучше понять модель.

Для добавления примечания на диаграмму можно использовать два инструмента. К элементу диаграммы можно прикрепить примечание-комментарий (note). Если же требуется прокомментировать целую диаграмму, применяется текстовая область (text box). В частности, в такой области указывается заголовок диаграммы.

На рис. 2.68 подпись «Пограничный класс является примечанием».

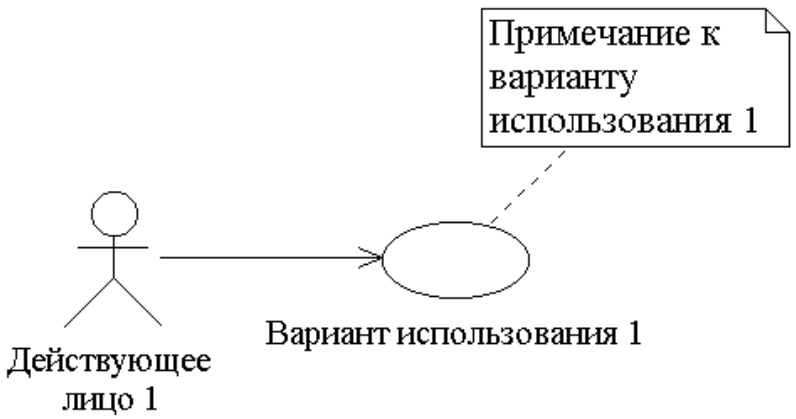


Рис. 2.68. Прикрепление примечания к варианту использования

На языке UML такие элементы, как действующие лица, варианты использования, классы и компоненты, можно сгруппировать в ***пакеты*** (packages). Это позволяет упорядочить элементы модели. Пакет изображается с помощью нотации показанной на рис. 2.69.

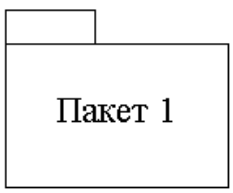


Рис. 2.69. Графическая нотация для изображения пакетов

2.8. Модели и ракурсы

Невозможно охватить все тонкие детали сложной программной системы одним взглядом. Необходимо понять как функциональные, так и структурные свойства системы. Следует уяснить также таксономическую структуру классов, используемые механизмы наследования, индивидуальное поведение объектов и динамическое поведение системы в целом. На рис. 2.70 представлены различные типы моделей, которые считаются главными в объектно-ориентированном подходе. Через них будут выражаться результаты анализа и проектирования, выполненные в рамках любого проекта. Эти модели в совокупности семантически достаточно богаты и универсальны, чтобы разработчик мог выразить все заслуживающие внимания стратегические и тактические решения, которые он должен принять при анализе системы и формировании ее архитектуры. Кроме того, эти модели достаточно полны, чтобы служить техническим проектом реализации на любом объектно-ориентированном языке программирования.

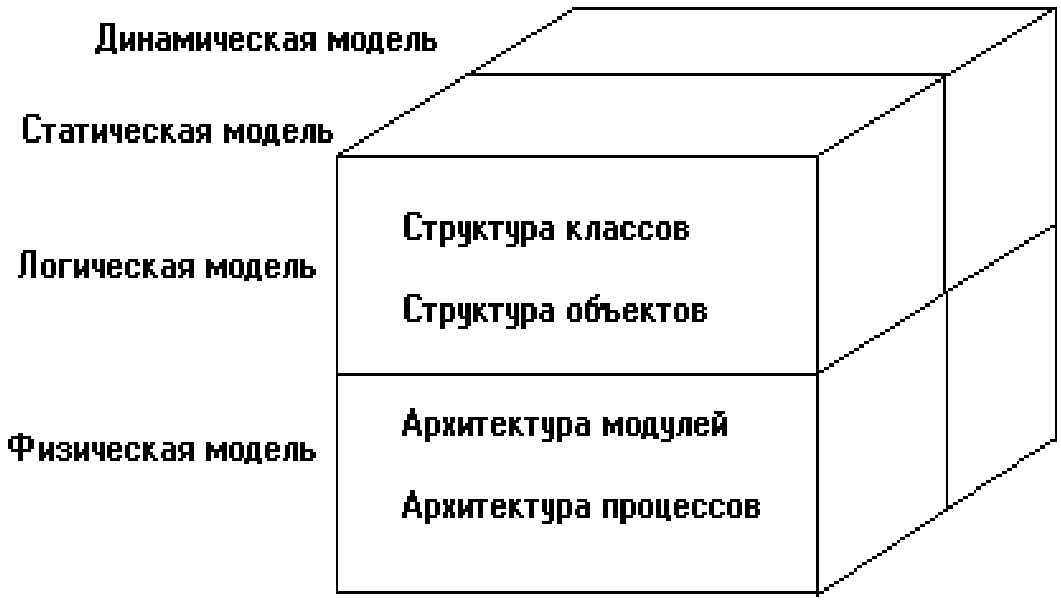


Рис. 2.70. Объектные модели

При принятии решений в анализе и проектировании полезно рассмотреть взаимодействие классов и объектов в двух измерениях: логическом/физическом и статическом/динамическом (рис. 2.70). Оба аспекта

необходимы для определения структуры и поведения объектной системы.

В каждом из двух измерений строят несколько диаграмм, которые представляют систему в различных ракурсах. Диаграммы содержат информацию о ключевых абстракциях системы, их связях и поведении. В установившемся состоянии проекта все диаграммы должны быть согласованы между собой и со всей моделью.

1. Логическая и физическая модели. ***Логическое представление*** описывает перечень и смысл ключевых абстракций и механизмов, которые формируют предметную область или определяют архитектуру системы.

Физическая модель определяет конкретную программно-аппаратную платформу, на которой реализована система.

При анализе мы должны задать следующие вопросы. Каково требуемое поведение системы? Каковы роли и обязанности объектов по поддержанию этого поведения?

При проектировании необходимо задать следующие вопросы относительно архитектуры системы.

1. Какие существуют классы, и какие есть между ними связи?
2. Какие механизмы регулируют взаимодействие классов?
3. Где должен быть объявлен каждый класс?
4. Как распределить процессы по процессорам и как организовать работу каждого процессора, если требуется обработка нескольких процессов?

Чтобы ответить на эти вопросы, используются следующие диаграммы: Диаграммы Вариантов Использования; Кооперативные диаграммы; Диаграммы Классов; Диаграммы Компонентов; Диаграммы Размещения.

2. Статическая и динамическая модели. Перечисленные в предыдущем пункте типы диаграмм являются в большей части ***статическими***. Но практически во всех системах происходят события: объекты рождаются и уничтожаются, посылают друг другу сообщения (причем в определенном порядке), внешние события вызывают операции объектов. Описание ***динамических*** событий на статическом носителе, например, на листе бумаги, будет трудной задачей, но эта же трудность встречается фактически во всех областях науки. В объектно-ориентированном проектировании мы отражаем динамическую семантику двумя дополнительными диаграммами: Диаграммы Последовательности; Диаграммы Состояний. Каждый класс может иметь собственную Диаграмму Состояний, которая показывает, как объект класса переходит из состояния в состояние под воздействием событий. На Диаграмме Последовательности можно показать порядок передачи сообщений.