Предисловие

Учебное пособие предназначено для студентов IV курса факультета информатики и вычислительной техники ЯрГУ. Пособие содержит описание основных элементов графической нотации языка UML, определения ключевых терминов, их семантику и особенности графического изображения на различных диаграммах. В пособии описаны основы методологии объектно-ориентированного анализа и проектирования, рассмотрены канонические диаграммы языка UML и приведены примеры реализации отдельных элементов графической нотации на языке C++.

В первой главе пособия описываются основные понятия методологии объектно-ориентированного анализа и проектирования, определение языка UML, его структура и история развития.

Вторая, третья и четвертая главы учебного пособия посвящены описанию процесса разработки логического представления модели сложной системы, которое включает в себя совокупность построения моделей трех видов: модели классов, состояний и взаимодействий. Каждая из трех глав содержит семантику и графическую нотацию канонических диаграмм языка UML, необходимых для проектирования соответствующих моделей. Здесь также обсуждаются вопросы, связанные с реализацией конструкций языка UML на языке объектно-ориентированного программирования C++.

Пятая глава посвящена завершающему этапу построения модели сложной системы, а именно её физическому представлению. В данной главе рассматриваются правила проектирования диаграмм реализации, которыми являются диаграммы компонентов и диаграмма размещения.

Материал, изложенный в пособии, снабжен большим количеством примеров и рисунков. Это позволяет лучше понять и освоить основы языка UML, а так же получить навыки разработки объектно-ориентированных моделей с последующей их реализацией на языке C++.

Учебное пособие содержит список литературы, рекомендуемой для тех, кто хочет освоить язык UML и научиться использовать его в процессе разработки программных приложений.

Введение

Компьютерные и информационные технологии без преувеличения можно назвать наиболее динамично развивающейся областью знаний, которые сосредотачивают в себе самые последние достижения в сфере науки и техники. Тенденции развития информационных технологий определяют постоянное возрастание сложности программного обеспечения, которая обусловлена следующими факторами:

- сложность реальной предметной области, из которой исходит заказ на разработку;
- сложность описания (большое количество функций, процессов, элементов данных, сложные взаимосвязи между ними), требующая тщательного моделирования и анализа процессов;
- отсутствие полных аналогов, ограничивающее возможность использования каких либо типовых проектных решений и прикладных систем;
- необходимость интеграции уже существующих и вновь разрабатываемых приложений;
- функционирование в неоднородной среде на нескольких аппаратных платформах;
- необходимость поддержания единого стиля для различных версий программ при их постоянной доработке и модификации в процессе разработки и эксплуатации в соответствии с изменяющимися потребностями заказчика или пользователя;
- разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств.

В настоящий момент сложность является существенным и неотъемлемым свойством современного программного обеспечения. Многие проблемы разработки информационных систем следуют из этой сложности и ее роста при увеличении размера проектируемого приложения, поэтому проектирование модели программной системы на стадии, предшествующей ее реализации или обновлению, в такой же мере необходимо, как и наличие проекта для строительства большого здания.

Наряду с этим, процесс создания современных программных приложений подразумевает участие в их разработке множества специалистов различной квалификации, для которых единообразное понимание архитектуры и функциональности разрабатываемых систем является серьезной проблемой. Следовательно, построение предварительной модели приложения до начала написания соответствующего программного кода становится настоящей необходимостью. Основным требованием к такой модели является то, что модель должна быть понятна как заказчику, так и всем специалистам проектной группы, включая программистов и удобна для документирования.

Все эти факторы привели к необходимости перехода от «кустарных» к «индустриальным» способам создания программных систем и появлению

совокупности инженерных методов и средств создания программного обеспечения, объединенных общим названием "*программная инженерия" (software engineering).*

В основе программной инженерии лежит идея о том, что проектирование программного обеспечения является формальным процессом, который можно изучать и совершенствовать, а освоение и правильное применение методов и средств создания программных приложений позволяет повысить их качество, обеспечить управляемость процесса разработки и увеличить срок жизни программных продуктов.

В истории становления и развития программной инженерии можно выделить два этапа:

- 70-е и 80-е гг. – систематизация и стандартизация процессов создания программного обеспечения на основе структурного подхода;
- 90-е гг. – переход к объектно-ориентированному подходу.

Сущность структурного подхода состоит в декомпозиции или разделении системы на небольшие подсистемы, каждую из которых можно разрабатывать независимо от других.

Структурные методы при разработке программного обеспечения широко применялись в 70-80-х гг. Однако, в связи с все возрастающей сложностью программных систем, моделирование которых структурный поход был уже не в состоянии обеспечить, и появлением объектно-ориентированных языков программирования, которые структурный подход не поддерживает, в 80-90-е гг. все большую популярность приобретает объектно-ориентированная методология.

Этот подход к разработке моделей сложных программных систем основан на представлении о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект как экземпляр определенного класса.

Объектно-ориентированный подход внес достаточно радикальные изменения в сами принципы создания и функционирования программ и, в то же время, позволил существенно повысить производительность труда программистов, по-иному взглянуть на проблемы и методы их решения, сделать программы более компактными и легко расширяемыми.

1. Основные понятия моделирования систем и программных приложений

Разработка модели сложной системы или программного приложения, как правило, предшествует их созданию или обновлению. Это необходимо для того, чтобы яснее представить себе решаемую задачу, так как невозможно «окинуть одним взглядом» сложную систему. Продуманные модели очень важны и для взаимодействия внутри команды разработчиков, и для взаимопонимания с заказчиком.

В основе современного подхода к моделированию сложных систем лежит принцип ***объектно-ориентированного анализа и проектирования (ООАП)***. Это методология разработки программных систем, в основу которых положена объектно-ориентированная концепция построения моделей предметной области в форме классов, обладающих структурными свойствами и поведением.

1.1. Основные понятия методологии ООАП

Фундаментальными понятиями ООАП являются понятия класса и объекта, а также ее основные принципы: абстракция, инкапсуляция, полиморфизм, наследование.

Класс (class) представляет собой *абстракцию* совокупности реальных *объектов*, которые имеют общий набор свойств и обладают одинаковым поведением.

Объект (object) в контексте ООАП рассматривается как *экземпляр соответствующего класса*.

Определение классов и объектов – одна из самых сложных задач объектно-ориентированного анализа и проектирования. Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Итак, объект обладает состоянием, поведением и идентичностью. Структура и поведение схожих объектов определяет общий для них класс. Термины «экземпляр класса» и «объект» взаимозаменяемы. Объекты, которые не имеют идентичных свойств или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

Абстракция (abstraction) – характеристика сущности, которая отличает ее от других сущностей. Другими словами она означает сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Сначала принимается решение о том, что представляет собой объект и что он делает, а уже затем подбирается способ его реализации.

Инкапсуляция (encapsulation) – сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к ним объектов или пользователей. Иначе говоря, сокрытие информации, состоит в отделении внешних аспектов объекта, доступных другим объектам, от деталей внутренней реализации, которые от других объектов скрываются. Таким образом, инкапсуляция исключает возникновение взаимосвязей участков программы, из-за которых небольшие изменения приводят к значительным непредвиденным

последствиям. Инкапсуляция позволяет изменить реализацию объекта безо всяких последствий для использующих его приложений.

Полиморфизм (polymorphism) – свойство одноименных методов или операций выполнять разные действия или обладать различным поведением в зависимости от того, к какому классу они относятся. Это значит, что одна и та же операция подразумевает разное поведение в разных классах. При вызове операции не нужно беспокоиться о том, сколько реализаций этой операции существует в системе. Полиморфизм перекладывает ответственность за выбор подходящей реализации с вызывающего кода на иерархию классов. Каждый объект выбирает подходящую процедуру согласно своему классу. Это облегчает поддержку программ, потому что добавление нового класса не требует изменения вызывающего кода.

Наследование (inheritance) – принцип, в соответствии с которым знание о более общей категории разрешается применять для частной. То есть, если родительский класс обладает фиксированным набором свойств и поведения, то любой наследуемый от него класс (потомок) должен содержать тот же набор свойств и поведения, а возможно и иметь дополнительные, которые будут характеризовать его уникальность. Наследование структур данных вместе с поведением дает возможность подклассам совместно использовать общий код.

На использовании вышеперечисленных принципов основывается методология построения объектно-ориентированных моделей.

Модель (model) – абстракция произвольной системы или объекта, рассматриваемая с определенной точки зрения, и представленная на некотором языке или в графической форме. Попросту говоря, она является упрощенным представлением реальности. Хорошая модель должна описывать важнейшие аспекты проблемы и опускать все прочие.

Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. Модель состоит из множества элементов, которые совместно описывают моделируемую систему. Основное требование к модели – модель должна быть понятна всем специалистам проектной группы и удобна для документирования.

Процесс построения и последующего применения моделей называется ***моделированием.***

Сложную систему можно представить в виде совокупности моделей трех типов: модели классов, модели состояний и модели взаимодействий. Каждая из этих моделей описывает определенный уровень функционирования рассматриваемой системы. Другими словами, каждая модель соответствует некоторой определенной точке зрения на проектируемую систему.

Модель классов описывает объекты, входящие в состав системы и отношения между ними. Модель классов изображается на диаграммах классов. Диаграммы показывают общую структуру и поведение, а также связи между классами.

Модель состояний описывает историю жизни объектов. Модель состояний изображается на диаграммах состояний. Каждая диаграмма показывает порядок состояний и событий, возможный в рамках данной системы для одного класса объектов. Действия и события на диаграмме состояний становятся операциями объектов модели классов. Ссылки между диаграммами состояний становятся взаимодействиями в модели взаимодействия.

Модель взаимодействий описывает взаимодействия между объектами. Модели состояния и взаимодействия описывают разные аспекты поведения. Модель взаимодействия изображается с помощью вариантов использования и на диаграммах последовательности и деятельности.

Три описанные модели являются связанными между собой составляющими полного описания системы. Несмотря на то, что каждая модель описывает свои аспекты системы, она ссылается на другие модели. Модель классов описывает структуры данных, которыми оперируют модели состояний и взаимодействия. Операции в модели классов связаны с событиями и действиями. Модель состояний описывает структуру управления объектов.

Появление объектно-ориентированной методологии разработки моделей сложных систем в свою очередь потребовало удобного инструмента для моделирования, визуального представления и описания проектируемых программных приложений.

Однако оказалось, что разработка соответствующего языка, который объединял бы сильные стороны известных методов и обеспечивал наилучшую поддержку моделирования, является непростым делом. Потребовалось несколько лет, прежде чем усилия группы специалистов ведущих фирм производителей привели к появлению современной версии языка UML.

1.2. История развития языка UML

Мощный толчок к разработке этого направления информационных технологий дало распространение объектно-ориентированных языков программирования в конце 80-х начале 90-х годов прошлого века. Пользователям хотелось получить единый язык моделирования, который объединил бы в себе всю мощь объектно-ориентированного подхода и давал бы четкую модель системы, отражающую все ее значимые стороны.

К середине 90-х годов лидерами в этой области стали методы Booch (Грайди Буч), OMT (Object Modeling Technique) (Джим Рамбо), OOSE – (Obgiect Oriented Software Engineering) (Ивар Якобсон). Однако эти три метода имели свои сильные и слабые стороны:

- OOSE был лучшим на стадии анализа проблемной области и анализа требований к системе;
- OMT – наиболее предпочтителен на стадии анализа и разработки информационных систем, ориентированных на обработку больших объемов данных;
- Booch лучше подходил для этапов проектирования и конструирования модели.

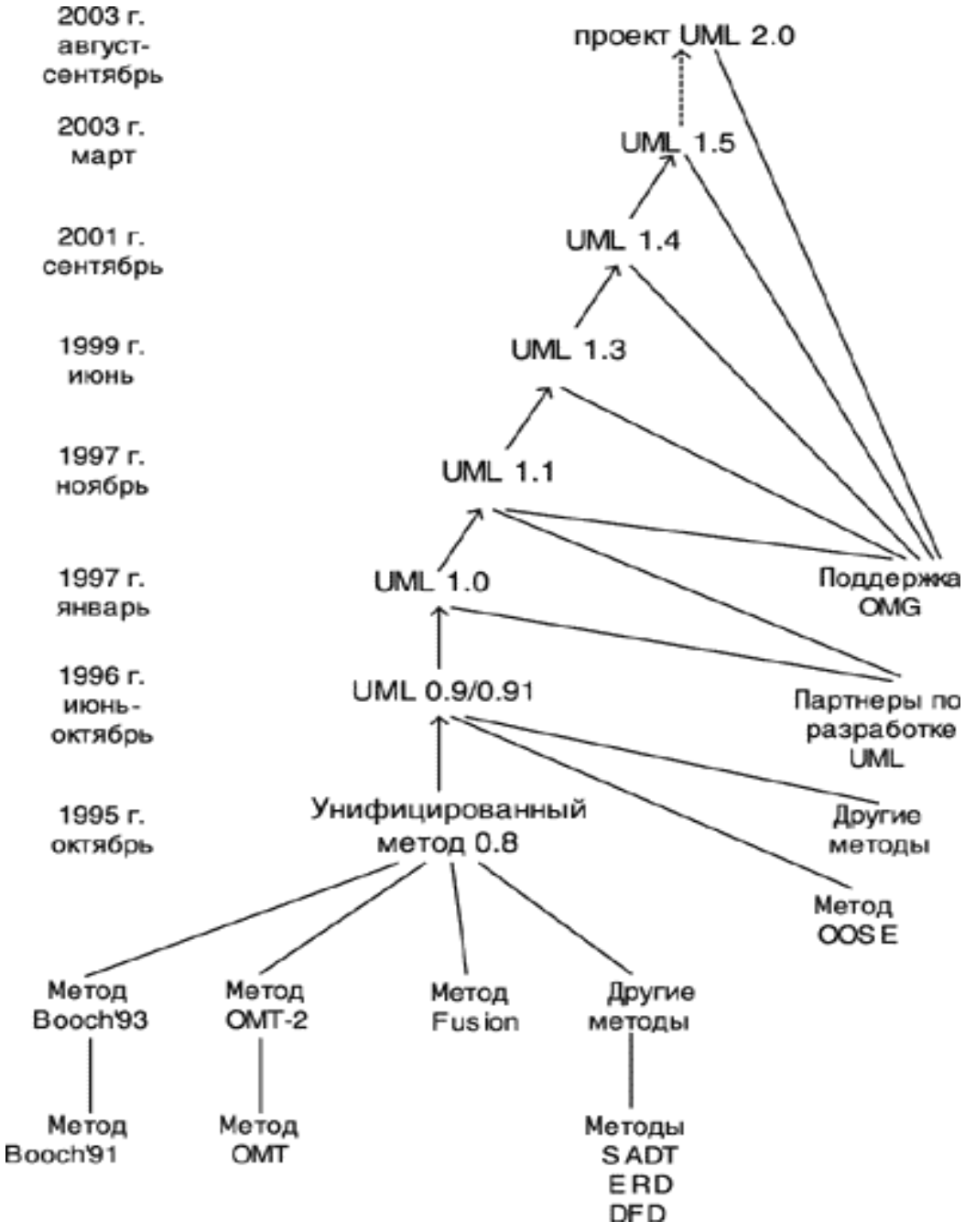


Рис. 1.1. Развитие языка UML

История развития языка UML (рис.1.1) берет начало с 1994 года, когда Джим Рамбо и Грайди Буч начали работу над объединением систем обозначений OMT и Буча. Осенью 1995 года увидела свет первая черновая

версия объединенной методологии, которую они назвали Unified Method 0.8. В 1995 году к ним присоединился Ивар Якобсон, который принес с собой концепцию OOSE.

В 1996 году группа управления объектами (Object Management Group – OMG) объявила конкурс на лучший стандарт обозначений для объектно-ориентированного моделирования. В этом конкурсе приняло участие несколько компаний. В результате их предложения были объединены в единую систему. Окончательной доработкой стандарта занялась фирма Rational. В этом процессе активное участие принимали Буч, Рамбо и Якобсон. В результате в январе 1997 года появилась первая версия языка UML, которая была достаточно хорошо определена и обеспечивала решение широкого класса задач, а в ноябре 1997 года язык UML был принят группой OMG в качестве стандарта.

Созданная система обозначений UML оказалась настолько удачной, что со временем вытеснила практически все другие системы, а язык UML был принят в качестве стандартного языка моделирования и используется сейчас практически всеми крупнейшими компаниями – производителями программного обеспечения.

В настоящее время интерес к языку UML со стороны специалистов в различных областях все более возрастает, так как заложенные в унифицированном языке моделирования возможности могут быть использованы как для объектно-ориентированного анализа и проектирования программных систем, так и для документирования бизнес-процессов, а в более широком контексте – для представления знаний в интеллектуальных системах и сложных программно-технологических комплексах.

1.3. Определение языка UML

Язык UML (Unified Modeling Language) или унифицированный язык моделирования – предназначен для описания, визуализации, проектирования и документирования объектно-ориентированных систем и бизнес-процессов с ориентацией на их последующую реализацию в виде программного обеспечения.

Итак, UML – это прежде всего спецификация, то есть подробное описание системы, которое полностью определяет ее цель и функциональные возможности.

Не следует забывать, что заказчик и разработчик являются, как правило, специалистами в абсолютно разных областях. Каждый из них понимает спецификации по-своему: постановка задачи, техническое задание, требования пользователя, архитектура системы и так далее. Другими словами, они «говорят на разных языках», зачастую не понимая друг друга. Вот поэтому и необходимо унифицированное средство создания спецификаций, достаточно простое и понятное для всех заинтересованных лиц.

Наряду с этим, UML является средством визуализации, то есть наглядного представления разрабатываемой системы в виде специальных графических конструкций, получивших название диаграмм.

Известным является тот факт, что изучение и понимание чего-то нового идет гораздо лучше, если документ содержит не только текст, но и иллюстрацию к нему. Проще говоря, картинки с подписями наглядны и понятны, причем они, как правило, однозначно воспринимаются всеми заинтересованными лицами и могут быть использованы в качестве средства общения между людьми. Графические средства унифицированного языка моделирования как раз и позволяют создавать такие простые и понятные картинки или диаграммы, описывающие рассматриваемую систему с разных сторон, и которые можно продемонстрировать и обсудить с заказчиком.

С помощью таких диаграмм, описывающих систему с разных точек зрения, язык UML позволяет строить или проектировать модели программных приложений. Такие UML - модели сами по себе уже являются документацией к разрабатываемой программной системе, причем документацией понятной даже для неспециалиста.

Таким образом, унифицированный язык моделирования или UML является достаточно мощным средством объектно-ориентированного проектирования, которое может быть эффективно использовано для построения моделей сложных систем различного целевого назначения.

Следует отметить, что термин «унифицированный» в названии языка UML не является случайным и имеет два аспекта. С одной стороны, он фактически устраняет многие из различий, которые существуют между известными ранее языками моделирования и методиками построения диаграмм. С другой стороны, создает предпосылки для унификации различных моделей и этапов их разработки для широкого класса систем, не только программного обеспечения, но и бизнес-процессов.

1.4. Общая структура языка UML

Форма представления моделей тесно связана с исходными целями моделирования. В процессе разработки программных систем и приложений наибольшее распространение получили визуальные модели, которые используют для представления своих элементов специальную графическую нотацию, используемую в языке UML.

Визуальное проектирование в UML можно представить как некоторый процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели системы к логической, а затем и к физической модели соответствующей системы. Любая задача, таким образом, моделируется при помощи некоторого набора иерархических диаграмм, каждая из которых представляет собой некоторую проекцию системы.

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей: семантики и нотации.

Семантика (semantics) – система правил и соглашений, определяющих толкование и придание смысла конструкциям некоторого языка.

Нотация (notation) – система условных обозначений, принятая в некотором языке для изображения и визуализации. Другими словами, нотация – это то, что в других языках называют «синтаксисом». Само слово «нотация» подчеркивает, что UML – язык графический и модели «не записывают», а рисуют.

Семантика определяется для двух категорий объектных моделей: структурных (статических) моделей и моделей поведения (динамических моделей).

Структурные модели (structured models) – модели, предназначенные для описания статической структуры сущностей или элементов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения.

Модели поведения (behavioral models) – модели, предназначенные для описания процесса функционирования элементов системы, включая их методы и взаимодействие между ними, а также процесс изменения состояний отдельных элементов и системы в целом.

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Диаграмма (diagram) – графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика.

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой ***графическую нотацию языка UML.***

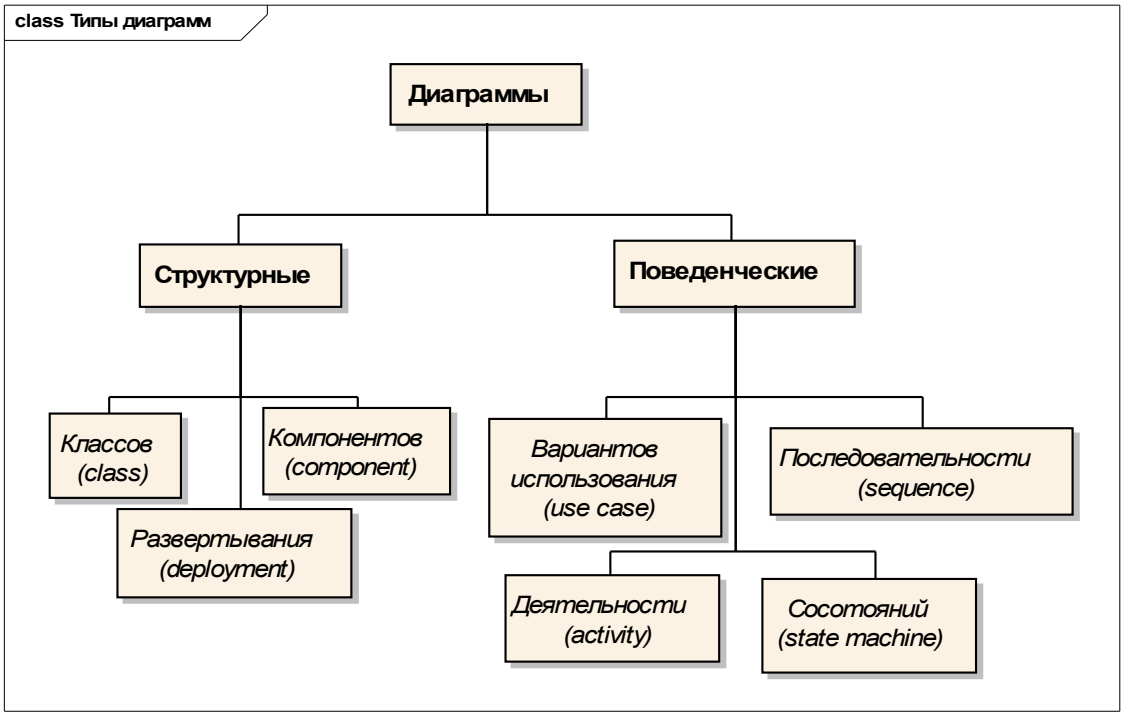


Рис. 1.2. Типы диаграмм языка UML

Представленные на рисунке 1.2 типы диаграмм на настоящий момент не составляют полный перечень всех видов диаграмм, применяемых для проектирования с использованием языка UML.

В связи с тем, что объектно-ориентированный подход приобретает все большую популярность и применяется для решения все более сложных задач, возникает необходимость в расширении средств проектирования, используемых в языке UML. То есть в процессе развития унифицированного языка моделирования в его нотации появляются новые виды диаграмм и новые элементы их визуализации. Однако следует отметить, что перечень диаграмм, используемых при разработке проекта, не является фиксированным и определяется самим разработчиком в зависимости от специфики задачи. Для простых приложений нет необходимости строить все без исключения диаграммы. Например, для локального приложения не обязательно строить диаграмму развертывания, отображающую инфраструктуру, на которую будет развернуто данное приложение.

Поэтому в рамках данного курса будут рассмотрены основные канонические диаграммы языка UML, которые наиболее часто используются для описания и проектирования различных систем. Это диаграммы:

- вариантов использования (use case diagram)
- классов (class diagram)
- последовательности (sequence diagram)
- состояний (statechart diagram)
- деятельности (activity diagram)
- компонентов (component diagram)
- развертывания (deployment diagram)

Большинство перечисленных выше диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, носит топологический характер, ни геометрические размеры, ни расположение элементов диаграмм не имеют принципиального значения. Проще говоря, UML предоставляет исключительную свободу – можно рисовать, что угодно и как вздумается, лишь бы можно было понять смысл созданных диаграмм.

Однако следует отметить, что, несмотря на всю свободу действий, которую предоставляет язык UML, при проектировании диаграмм нужно учитывать, что вся информация о моделируемой системе должна быть явно представлена на диаграммах. При этом диаграммы не должны содержать противоречий, их не следует перегружать текстовой информацией и каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов

Совокупность построенных диаграмм должна содержать всю информацию, которая необходима для реализации проекта системы любой сложности. Другими словами, диаграммы рисуют для визуализации системы с разных точек зрения, то есть каждая диаграмма сама по себе является лишь одной из проекций системы, поэтому только набор диаграмм составляет модель проектируемой системы и наиболее полно ее описывает.

Диаграммы в нотации языка UML изображаются в виде прямоугольной рамки или фрейма, который имеет область содержания и заголовок.

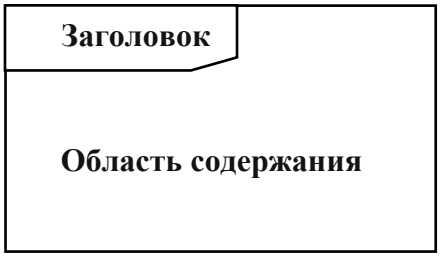


Рис. 1.3. Изображение диаграммы UML в виде фрейма.

Область содержания предназначена для изображения графических узлов и путей между ними, которые собственно и представляют собой элементы модели в нотации языка UML.

Заголовок диаграммы является строкой текста, записанной в прямоугольнике с обрезанным углом в верхнем левом углу фрейма и имеющей следующий синтаксис:

[<тип диаграммы>]<имя>[<параметры>]

В качестве типа диаграммы для заголовка, как правило, используются следующие сокращения:

- ***activity <act>*** - для фреймов диаграмм деятельности
- ***class*** - для фреймов диаграмм классов
- ***component <cmp>*** - для фреймов диаграммы компонентов
- ***sequence <sd>*** - для фреймов диаграмм последовательности
- ***package <pkg>*** - для фреймов диаграммы пакетов
- ***state machine <stm>*** - для фреймов диаграмм состояний
- ***use case <uc>*** - для фреймов диаграммы вариантов использования.

В заголовке также указываются имя и параметры соответствующего пространства имен или элемента модели, который владеет элементами модели в области содержания диаграммы. Если фрейм диаграммы опущен, заголовок также опускается.

Диаграммы, отражающие систему с различных точек зрения, можно и нужно строить в некоторой логической последовательности. Эта последовательность разработки модели проектируемой системы формируется

по мере приобретения опыта использования UML и понимания его конструкции. Кому-то удобнее начинать с построения диаграммы классов, а кому-то – с диаграммы вариантов использования, кому-то не нужны диаграммы объектов, а кто-то предпочитает диаграммам последовательности диаграммы кооперации.

Таким образом, последовательность построения диаграмм отражает персональный стиль проектирования и вырабатывается постепенно с приобретением опыта моделирования.

2. Моделирование классов

Центральное место в методологии объектно-ориентированного анализа и проектирования занимает разработка логической модели системы в виде *модели классов*.

Модель классов отражает различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. Другими словами, сначала нужно определить, что именно изменяется, а затем описывать когда и как это происходит. С этой точки зрения модель классов может служить основой дальнейшего развития модели проектируемой системы, так как она создает контекст для моделей состояний и взаимодействия. Поэтому от умения правильно выбрать классы и установить между ними взаимосвязи часто зависит не только успех процесса проектирования, но и производительность выполнения программы.

Цель конструирования модели классов состоит в том, что бы охватить те реальные концепции, которые важны для проектируемого приложения. Проще говоря, чтобы хорошо понять систему, лучше всего сначала изучить ее статическую структуру, то есть структуру объектов и отношений между ними в фиксированный момент времени, что и отражает модель классов.

Модель классов в языке UML изображается с помощью диаграмм классов.

2.1. Диаграммы классов

Процесс построения диаграммы классов является основополагающим при разработке проектов сложных объектно-ориентированных систем

Диаграмма классов (class diagram) — диаграмма языка UML, на которой представлена совокупность статических элементов модели, таких как *классы* с *атрибутами* и *операциями*, а также связывающие их отношения.

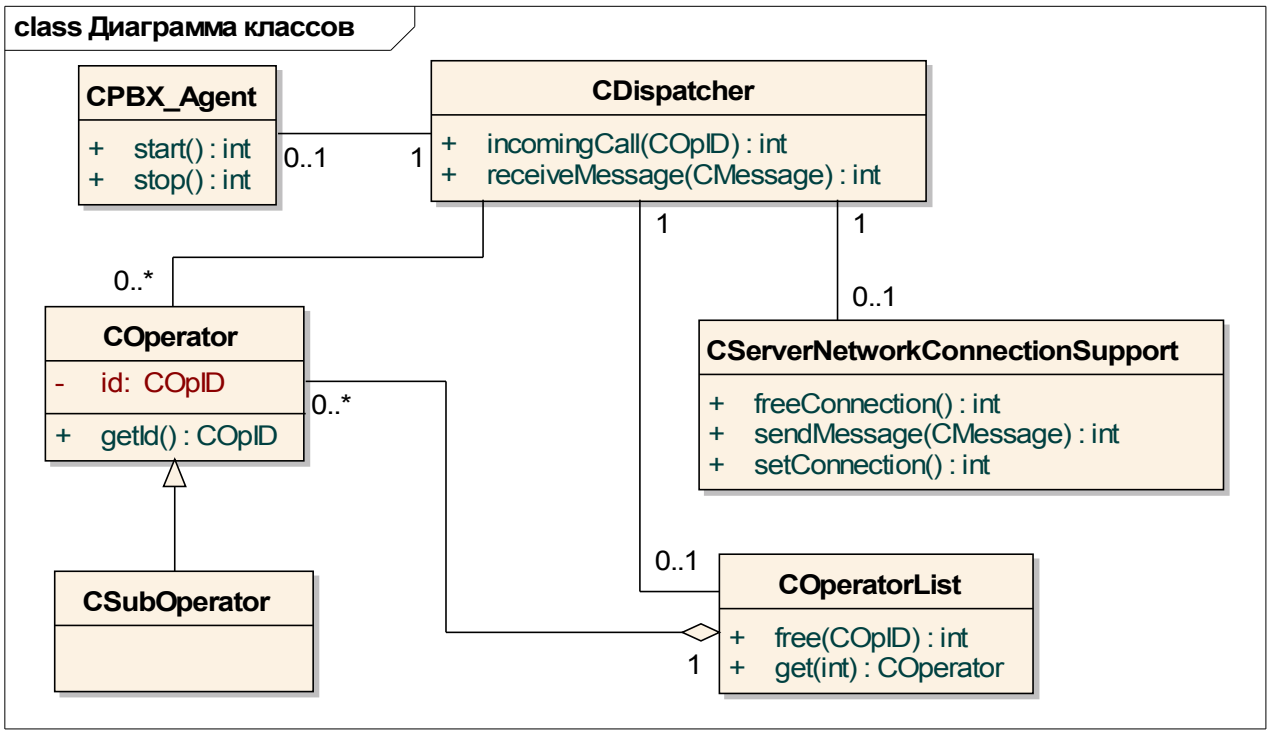


Рис. 2.1. Пример диаграммы классов (телефонная служба приема заявок, фрагмент)

Диаграммы классов позволяют описать модель классов и их отношений при помощи графической системы обозначений. Они полезны как для абстрактного моделирования, так и для проектирования конкретных программ. Диаграммы классов – один из наиболее часто используемых видов диаграмм UML.

Разработка диаграмм классов преследует следующие цели:

- определить сущности предметной области (объекты) и представить их в форме классов с соответствующими атрибутами операциями;
- определить взаимосвязи между сущностями предметной области и представить их в форме типовых отношений между классами.

На диаграммах классов изображаются классы с атрибутами и типами атрибутов, методами, их типами и параметрами, взаимосвязи между классами, а также иерархия наследования классов.

2.1.1. Классы и объекты

Объект (object) – это концепция, абстракция или сущность, обладающая индивидуальностью и имеющая смысл в рамках разрабатываемой модели. Каждый объект обладает индивидуальностью, которая предполагает наличие у него уникального имени, отличающегося от имен других возможных объектов того же класса. Объект является отдельным экземпляром класса.

Класс (class) – описывает множество (группу) объектов с одинаковыми свойствами (атрибутами), одинаковым поведением (операциями), типами отношений и семантикой.

Для графического изображения класса в языке UML используется прямоугольник, который дополнительно может быть разделен горизонтальными линиями на три части. В этих секциях могут указываться имя класса, атрибуты и операции класса.

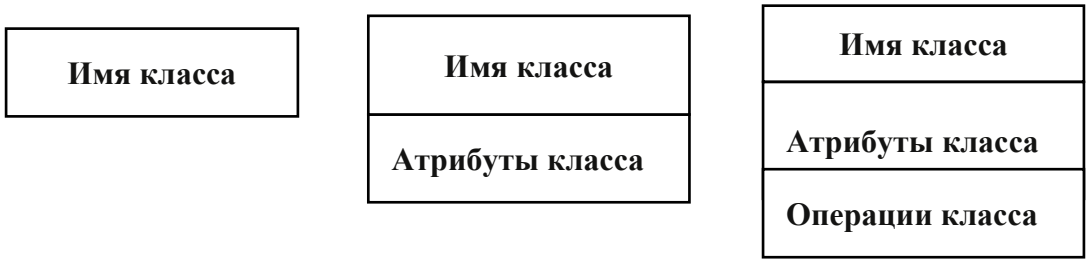


Рис. 2.2. Варианты изображения класса на диаграмме классов

Имя класса указывается в самой верхней секции прямоугольника и располагается по центру. Согласно требованиям языка UML имя класса должно записываться полужирным шрифтом, но это условие соблюдается крайне редко, так как многие автоматизированные средства разработки программного обеспечения (CASE-системы) не поддерживают данное правило, поэтому толщине шрифта можно не придавать принципиального значения.

В последнее время принято подразделять выполняемые проекты на два типа: с генерацией программного кода и без нее. Для проектов первого типа диаграмма классов служит моделью проектирования. Имена классов в этом случае следует записывать в соответствии с синтаксисом языка программирования, выбранного для генерации программного кода. Для проектов второго типа диаграмма классов служит моделью анализа. В этом случае для повышения наглядности разрабатываемой модели имена классов можно записывать символами кириллицы.

Имя класса должно быть уникальным и обязательно начинаться с заглавной буквы. При этом оно должно быть осмысленным и понятным и соответствовать тому объекту, который описывает данный класс.

Фирмы-разработчики программного обеспечения используют в начале имени класса определенную заглавную букву. Например, имя класса может начинаться с буквы «C», что означает «class». Такую форму записи имен классов использует фирма Microsoft. Имя класса также может начинаться с буквы «T». Такая форма записи используется фирмой Borland.

Если имена классов, состоят из нескольких слов, то имеет место соглашение о том, что подобные имена не должны содержать пробелов. Такие имена принято записывать с помощью конкатенации слов с использованием заглавных букв между строчных в начале каждого из слов, входящих в имя

класса. Например: *НомерСчета, CardReader*. Следует заметить, что данное соглашение не является требованием языка UML.

Имена, объединяющие в себе более трех слов, применять не рекомендуется. Такие длинные идентификаторы затрудняют чтение и понимание программы.

2.1.2. Атрибуты

Атрибут (attribute) – это именованное свойство класса, описывающее одно из значений, которое может иметь каждый объект класса. Иначе говоря, атрибут служит для представления отдельной характеристики или свойства, которое является общим для объектов данного класса. У каждого конкретного объекта атрибут принимает свое конкретное значение.

Не следует путать значения с объектами. Атрибут должен описывать значения, а не объекты.

В описании задач атрибут может быть именем существительным в описании (потоке) событий (некоторые из них будут классами или объектами, другие – действующими лицами, последняя группа – атрибуты) или прилагательным. Например: *«Пользователь вводит имя сотрудника, дату рождения, его адрес и номер телефона»*. Это значит, что у класса *Сотрудник* имеются атрибуты *Имя, Адрес, Дата рождения, Номер телефона.*

Согласно системе обозначений языка UML атрибуты класса записываются во второй сверху секции прямоугольника класса, которую называют секцией атрибутов. Все спецификации атрибутов выравниваются по левому краю.

Общий формат записи отдельного атрибута класса следующий:
<видимость> <имя атрибута> [кратность] : <тип> = <начальное значение>

Имя является единственным обязательным элементом в обозначении атрибута. Имя атрибута должно начинаться с маленькой буквы и не содержать пробелов. Оно уникально в рамках класса, но может быть одинаковым у разных классов.

После каждого атрибута могут быть указаны необязательные сведения о нем: кратность, тип, значение по умолчанию.

Так как атрибуты содержатся внутри класса, может понадобиться указать какие классы имеют право читать и изменять атрибуты. Для этого перед атрибутом указывается так же не обязательный модификатор или вид видимости, который защищает внутреннее устройство объекта путем ограничения доступа к атрибутам и операциям класса из других частей программы.

Этот параметр может принимать одно из четырех значений:

- ***public*** *(общий, открытый),* символ **“ + ”** – это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута.
- ***private*** *(закрытый, секретный),* символ **” - “** – соответствующий атрибут не виден никаким классом, а значит никакой другой класс не сможет его ни увидеть, ни редактировать.
- ***protected*** *(защищенный),* символ **” # ”** – такой атрибут доступен только самому классу и его потомкам.
- ***package*** *(пакетный),* символ **” ~ ”** – предполагает, что данный атрибут является общим, но только в пределах его пакета. То есть атрибут, имеющий пакетную видимость, может быть просмотрен и изменен другим классом, если этот класс находится в том же пакете.

Если видимость не указана - это означает, что она не определена. В общем случае атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код, так как модификатор видимости реализует принцип инкапсуляции, то есть защищает внутреннее устройство объекта путем ограничения доступа к атрибутам и операциям класса из других мест программы.

Кратность применительно к атрибутам характеризует общее количество конкретных значений для атрибута, которые могут быть заданы для объектов данного класса

Тип атрибута представляет собой имя некоторого типа данных, соответствующего основным типам данных того языка программирования, который предполагается использовать для реализации проектируемой модели. В общем случае тип атрибута записывается строкой текста, имеющей осмысленное значение в пределах модели, к которой относится данный класс.

Значение по умолчанию – это выражение, которое служит для задания начального значения данного атрибута в момент создания отдельного экземпляра соответствующего класса. Конкретное значение по умолчанию должно соответствовать типу данного атрибута. Если этот терм не указан, то значение атрибута на момент создания нового экземпляра класса не определено.

2.1.3. Операции

Операция (operation) – это функция или процедура, которая может быть применена к объектам класса. Операция служит для представления отдельной характеристики поведения, которая является общей для всех объектов данного класса, то есть все объекты одного класса имеют общий список операций.

Методом – называется реализация операции в данном классе.

Каждая операция в качестве неявного аргумента принимает свой целевой объект. Поведение операции зависит от класса целевого объекта. Объект всегда знает свой собственный класс, а потому он всегда знает и правильную реализацию операции. Одна и та же операция может быть применена к разным

классам. Такая операция называется *полиморфной*: в разных классах она может принимать разные формы. Система обозначений языка UML предписывает перечислять операции в третьем отделе прямоугольника класса.

Общий формат записи отдельной операции класса следующий:

<видимость> <имя операции> (список аргументов):<тип возвращаемого значения>.

Обязательной частью строки записи операции является наличие имени операции и круглых скобок. Имя операции указывается с маленькой буквы и выравнивается по левому краю. Оно должно быть уникальным для каждой операции данного класса и соответствовать тому действию, которое выполняет операция. Перед именем операции может быть указана видимость, семантика которой аналогична семантике видимости атрибутов. После имени операции могут быть указаны необязательные сведения: список аргументов и тип возвращаемого результата.

Список аргументов указывается в круглых скобках через запятую. Каждый из аргументов может быть представлен в следующем виде:

<направление> < имя аргумента> : <тип> = <значение по умолчанию>.

Перед именем аргумента может быть указан один из трех видов направления аргумента:

- ***in*** *(входной)* – указывает на то, что значения параметра передаются в операцию вызывающим объектом
- ***inout*** *(изменяемый)* – указывает на то, что значения параметра передаются в операцию вызывающим объектом и затем возвращаются обратно вызывающему объекту по окончании выполнения операции
- ***out*** *(выходной)* – указывает на то, что значения параметра передаются вызывающему объекту по окончании выполнения операции

Если направление операции не указано, то по умолчанию принимается in.

Семантика обозначения типа аргумента и значения по умолчанию аналогична правилам описания соответствующих термов для атрибутов.

Примеры изображения классов с атрибутами и операциями приведены на рис. 2.3.

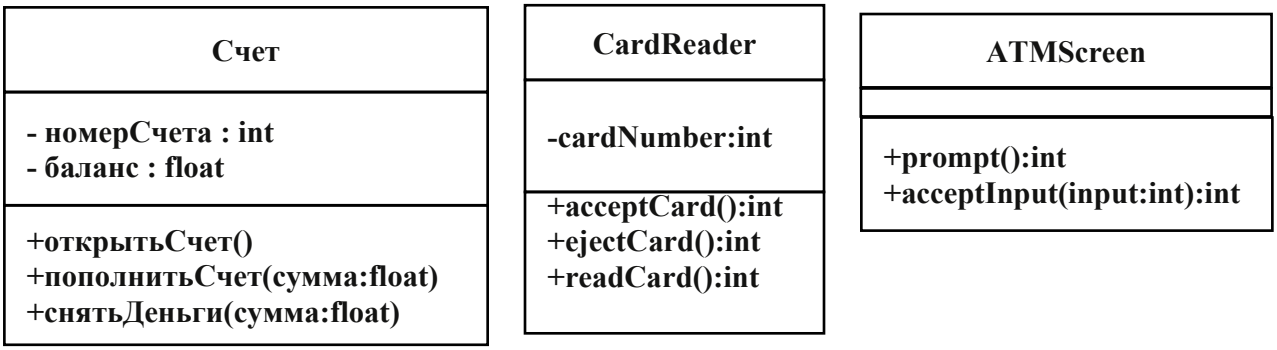


Рис. 2.3. Варианты изображения классов с атрибутами и операциями

Итак, класс обозначается прямоугольником, который может содержать до трех секций (отделов). В верхнем отделе указывается имя класса, которое является обязательным элементом. Во втором отделе – список атрибутов класса, в третьем – список операций. Отделы со списком атрибутов и операций являются необязательными элементами системы обозначений.

Следует обратить внимание, что отсутствие отдела атрибутов или операций говорит о том, что в данном представлении атрибуты не указаны. Напротив, наличие пустого отдела говорит о том, что атрибуты или операции отсутствуют.

2.2. Отношения между классами

Кроме внутреннего устройства классов важную роль при разработке моделей программных систем имеют отношения между классами, которые также могут быть изображены на диаграммах классов. Такими типами отношений являются отношения:

- ассоциации *(association)*
- обобщения *(generalization)*
- агрегации *(aggregation)*
- композиции *(composition)*

2.2.1. Отношение ассоциации

Ассоциация (association) – произвольное отношение или взаимосвязь между классами. Ассоциация описывает группы связей, обладающих общей структурой и семантикой. Другими словами, если два класса связаны друг с другом ассоциацией, то это означает, что их объекты (экземпляры) определенным образом связаны друг с другом. Например: вызывают методы друг друга, объекты одного класса являются параметрами (аргументами) методов другого, один класс имеет атрибут с типом другого класса (или указывает на него).

Система обозначений языка UML предписывает изображать ассоциацию сплошной линией, соединяющей между собой классы. В качестве дополнительных специальных символов, которые характеризуют специальные свойства ассоциации, могут использоваться символ навигации, имя ассоциации, имена и кратность концов ассоциации.

Наиболее простой и наиболее часто используемый случай данного отношения – *бинарная ассоциация (binary association)*, которая служит для представления произвольного отношения между двумя классами. Бинарные ассоциации могут быть однонаправленными и двунаправленными.

На однонаправленной ассоциации изображают только одну стрелку (символ навигации), показывающую ее направление (рис. 2.4а). Двунаправленные ассоциации рисуют в виде простой линии без стрелок (рис. 2.4в) или со стрелками, с обеих ее сторон (рис. 2.4б).

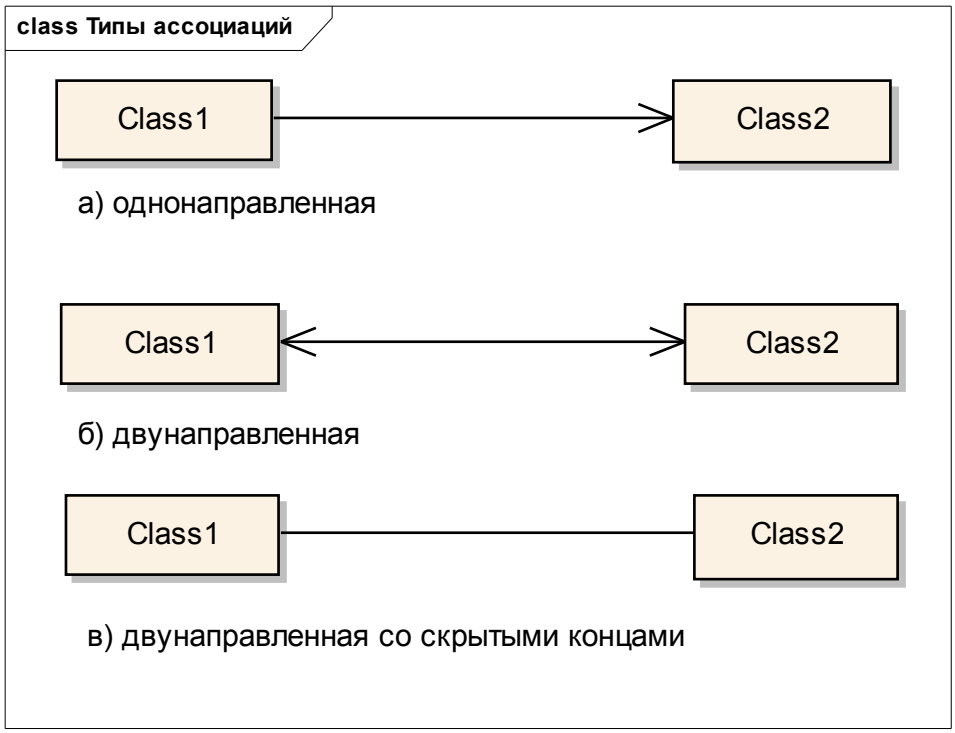


Рис. 2.4. Графическое изображение ассоциаций

Важно отметить, что при создании модели необходимо внимательно следить за направлением используемых ассоциаций и везде, где нет необходимости в поддержании связи в двух направлениях, использовать односторонние ассоциации. Это связано с тем, что односторонние ассоциации легче поддерживать, и они сокращают взаимную зависимость классов.

Существующие объектно-ориентированные языки и в частности C++ не предоставляют непосредственной поддержки концепции ассоциаций. Однако они позволяют с легкостью создавать связи в виде указателей или ссылок на объекты.

При реализации ассоциации следует помнить, что когда один класс ссылается на другой, последний должен быть виден и доступен исходному классу. При этом ссылающийся класс должен поддерживать ассоциацию своими силами. Поэтому, насколько возможно сократить зависимости, настолько облегчается обслуживание системы и повышается возможность повторного использования ее элементов.

Язык C++ позволяет реализовать ассоциации двумя способами: в виде атрибутов-указателей и в виде атрибутов-ссылок. Основное отличие между этими способами состоит в том, что указатель позволяет изменять связь или делать ее нулевой, тогда как ссылка получает свое значение в момент инициализации и не может быть изменена или сделана нулевой.

Как уже говорилось, доступ по ассоциации может быть двунаправленным и однонаправленным. Если ассоциация между классами C1 и C2 двунаправленная, то она может быть реализована с помощью атрибутов-указателей следующим образом:

```
class C1 {
   C2* a1;
. . . . . .
};

class C2 {
   C1* a2;
. . . . . .
};
```

Если ассоциация однонаправленная, например от класса C1 к классу C2, то это означает, что в программном коде класса C2 нет атрибута a2, и объекты класса C2 не знают, что на них ссылаются объекты класса C1. Например, если человек может менять работодателей или вообще не работать, то получится следующий код на языке C++:

```
class Company {
. . . . . .
};

class Person {
   Company* employer;
. . . . . .
};
```

Этот код допускает отсутствие работодателя. В этом случае указателю присваивается значение *null*.

Кроме того, ассоциации в языке C++ можно реализовывать и с помощью атрибутов-ссылок. Ссылка на класс подразумевает постоянную связь, в результате которой содержащий эту связь объект зависит от объекта атрибута. Ссылки должны быть связаны в момент инициализации. Они, в отличие от указателей, не могут быть нулевыми и не могут изменять свое значение. Таким образом, язык гарантирует наличие объекта ссылки. Например, если счет (Account) создается в конкретном банке (Bank) и не может быть передан в другой банк, и никакой счет не может существовать без банка, получиться следующий код:

```
class Bank {
. . . . . .
};

class Account {
   Bank& bank;
. . . . . .
};
```

Это требование языка C++ делает необходимым существование экземпляра класса Bank перед порождением экземпляра класса Account.

При использовании атрибутов-ссылок для реализации ассоциаций следует быть осторожным, чтобы не допустить непредвиденного изменения существующих в ассоциации объектов. Объект, содержащий атрибут-ссылку, может послужить «черным ходом» к другому объекту, на который он ссылается. Особенно важно следить за инкапсуляцией атрибутов-ссылок, которые должны изменяться и возвращаться только осмысленными и безопасными методами. C++ позволяет повысить уровень защищенности при помощи квалификатора *const*, который можно добавлять в тех случаях, когда изменение объекта в контексте ассоциации не предвидится.

Создание и удаление связей, соответствующих ассоциациям между классами, должно обеспечиваться методами, которые устанавливают или удаляют определенную связь в процессе своего выполнения.

Для установки связи, соответствующей односторонней ассоциации, объект может запоминать идентификатор другого объекта, являющегося параметром операции. Двусторонние связи могут быть созданы путем обмена идентификаторами. Выбор класса, управляющего обменом, определяется логикой разрабатываемого приложения.

Удаление связи - процедура, обратная ее созданию. В большинстве случаев это означает, что атрибут-указатель устанавливается равным нулю или из совокупности указателей удаляется один элемент.

В C++ создание и уничтожение связей осуществляется парой конструктор-деструктор. Эти связи обычно отражают выделение и освобождение ресурсов.

Важно отметить, что, несмотря на то, что ассоциации реализуются в программном коде в виде атрибутов, в процессе моделирования их следует изображать именно в виде символа ассоциации – направленной стрелки, соединяющей два класса. Ассоциации нельзя вносить в раздел атрибутов проектируемых классов. Это неприемлемо при моделировании, так как затрудняет понимание визуальной модели взаимодействия между классами.

Как уже говорилось, в качестве дополнительных (необязательных) символов могут использоваться: имя ассоциации, символ навигации, а так же имена, видимость и кратность концов ассоциации.

Имя ассоциации является необязательным элементом обозначения. Оно записывается с большой буквы рядом с линией ассоциации.

В качестве примера можно рассмотреть отношение между двумя классами – классом *Клиент* и классом *Счет*. Они связаны между собой бинарной ассоциацией с именем *ИмеетСчет* (рис. 2.5).

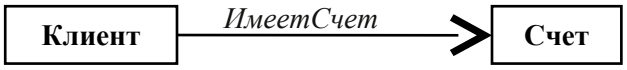


Рис. 2.5. Изображение ассоциации с именем

Имя ассоциации обычно используется, когда между двумя классами существует несколько ассоциаций. В этом случае каждой ассоциации дается свое имя для того, чтобы можно было отличать их между собой.

При изображении ассоциации особая роль отводится ***полюсам (концам) ассоциации***, которые графически соответствуют точке соединения линии ассоциации с прямоугольником класса. Именно у этих конструкций, соединяющих ассоциации с классами, определяются различные свойства, такие как *кратность* и *имя*.

Кратность (множественность) полюса ассоциации (multiplicity) – определяет возможное количество экземпляров одного класса, которые могут быть связаны с одним экземпляром другого класса через одну ассоциацию.

На диаграммах классов кратность указывается явно около конца линии, которой обозначается ассоциация. Значение кратности указывается в виде интервала целых чисел или константой. В языке UML приняты следующие соглашения для обозначения кратности (множественности):

кратность	**значение**
*	много
0..*	ноль или больше
1..*	один или больше
<число>	ровно число
<число1> .. <число2>	между числом1 и числом2
<число1>,<число2>	число1 или число2

Кратность «много» указывает, что объект связан с произвольным количеством объектов, однако для каждой ассоциации между конкретной парой объектов может существовать только одна связь. Значение кратности позволяет понять, является ли данная связь обязательной. Если между двумя объектами должно быть две связи, необходимо указать две ассоциации.

Определим кратность полюсов ассоциации для классов *Счет* и *Клиент* из предыдущего примера. Кратность «1» для класса Счет означает, что каждый счет соответствует только одному Клиенту. Кратность «1..*» для класса Клиент означает, что каждый клиент может иметь несколько счетов (рис. 2.6).

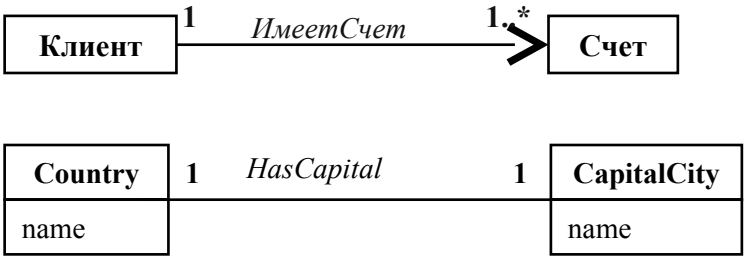


Рис. 2.6. Изображение кратности полюсов ассоциации

На рис. 2.6 приведен и другой пример кратности полюсов ассоциации: каждая *Страна* имеет одну *Столицу*, а *Столица* соответствует только одной *Стране*.

На начальных этапах разработки, как правило, не стоит беспокоиться о правильных значениях кратности. Сначала нужно определить классы и ассоциации, а затем указывать кратность. Если кратность не указана на диаграмме, то она считается неопределенной.

Следует отметить, что кратность полюсов ассоциации необходимо учитывать при написании исходного кода программы. Ассоциацию типа «один к одному» можно реализовать посредством взаимных указателей двух объектов. Ассоциация типа «один ко многим» требует одного указателя на одном полюсе и совокупности указателей на другом. Например, если каждый экземпляр товара Product относится только к одной последней продаже Sale, то каждый экземпляр Sale может указывать на совокупность проданных товаров.

```
class Product {
Sale* lastSale;
. . . . . .
};

class Sale {
Product** productSold;
. . . . . .
};
```

Во всех объектно-ориентированных языках и C++ в том числе имеются различные типы объектных совокупностей, которые можно использовать для реализации связи на полюсе с кратностью «много».

Полюс кроме кратности может иметь и свое собственное ***имя полюса ассоциации.*** Использование имен полюсов ассоциации не является обязательным, но чаще всего оказывается проще указывать имена полюсов вместо имен ассоциаций, или, по крайней мере, вместе с ними. Именами полюсов удобно пользоваться для прослеживания ассоциаций, чтобы делать спецификации более понятными и наглядными, так как имена полюсов ассоциации позволяют различать между собой разные ассоциации между одними и теми же классами.

Имя полюса пишется с маленькой буквы около полюса ассоциации рядом с соответствующим классом. Имя полюса ассоциации указывает на специфическую роль, которую играет класс, являющийся концом рассматриваемой ассоциации. Имя полюса ассоциации также может иметь видимость, которая определяет возможность доступа с других концов ассоциации. Способ записи видимости был рассмотрен выше.

В качестве примера рассмотрим ассоциацию между классами *Person* и *Company*. Объект класса *Person* может являться сотрудником для объекта

класса *Company*. Об этом говорит имя полюса ассоциации – "*employee"*. В свою очередь объект класса *Company* является работодателем для объектов класса *Person*, что и обозначает имя полюса ассоциации *"employer"*.

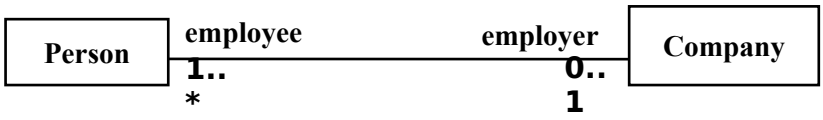


Рис. 2.7. Изображение ассоциации с именами и кратностью полюсов

Таким образом, каждый отдельный класс в ассоциации может играть определенную роль, которая и специфицируется на диаграмме классов с помощью имени полюса ассоциации.

Как уже говорилось, именовать полюса ассоциации не обязательно, но если ассоциация связывает класс с ним же самим, то в этом случае нужно именовать полюса ассоциации всегда. Такая ассоциация является частным случаем бинарной ассоциации и называется - ***рефлексивная ассоциация***, то есть это такая ассоциация, которая связывает класс с самим собой.

Примером такой ассоциации может служить класс *CListItem,* реализующий элемент двусвязного списка. У данного класса есть ассоциация с самим собой, которая указывает, что один объект этого класса связан с двумя другими объектами – с одним через полюс *prev* (предыдущий в списке), с другим через полюс *next* (следующий в списке). При этом экземпляр данного класса может быть связан только с одним экземпляром того же класса (если он первый или последний в списке) или не связан ни с одним вовсе (если он один в списке). Эти роли используются в качестве имен для концов ассоциации, а количество объектов, с которыми может быть связан экземпляр класса *CListItem* по этой ассоциации указаны в виде значений кратности около соответствующего полюса ассоциации (рис. 2.8).

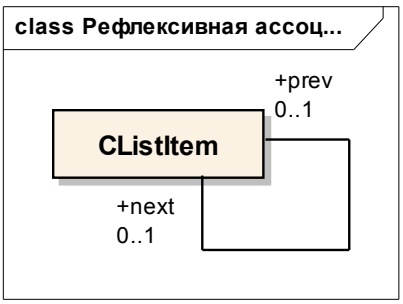


Рис. 2.8. Изображение рефлексивной ассоциации

Помимо имени и кратности полюс ассоциации с обозначением кратности «много» может иметь ещё и *строку свойства*. Строка свойства записывается в фигурных скобках и служит для указания дополнительных характеристик

соответствующего полюса ассоциации. В UML набор и назначение свойств полюса ассоциации определен следующим образом:

- *{bag}* – совокупность элементов, в которой допускается наличие дубликатов;
- *{sequence}*- упорядоченная совокупность, в которой также допускается наличие дубликатов;
- *{ordered}*- запрещает наличие дубликатов, упорядоченное множество.

Если около полюса указано слово {bag} или {sequence}, связей между двумя объектами может быть несколько. Если же этих указаний нет, связь может быть только одна.

Для примера с классами *Клиент* и *Счет* конец ассоциации может быть специфицирован с помощью строки свойства {ordered}, так как конкретные счета не могут повторяться, и являются упорядоченными, например, по дате их открытия (рис. 2.9).

Другой пример использования строки свойства: маршрут (класс *Itinerary*) это последовательность аэропортов (класс *Airport*), причем один и тот же аэропорт можно посетить несколько раз, то есть полюс ассоциации имеет свойство {sequence} (рис. 2.9), так как последовательность аэропортов является упорядоченной и допускает наличие повторений, например: Москва-Париж-Москва.

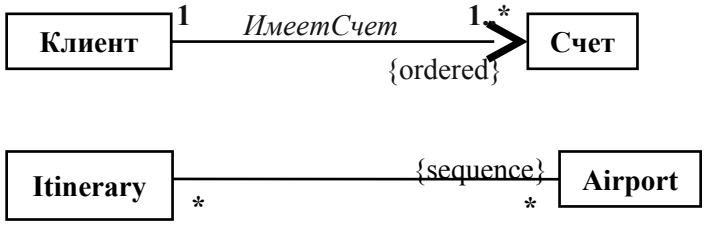


Рис. 2.9. Изображение ассоциации со строкой свойства

То есть, разница между {ordered} и {sequence} состоит в том, что первое обозначение строки свойства полюса ассоциации запрещает наличие дубликатов, а второе – разрешает.

2.2.2. N-арные ассоциации. Ассоциация-класс.

Кроме наиболее часто используемых бинарных ассоциаций в UML возможны и так называемые ***n-арные ассоциации***, которые связывают между собой три и более класса. Графически *n*-арная ассоциация обозначается ромбом, от которого ведут линии к символам классов, связанных данной ассоциацией. Имя *n*-арной ассоциации в этом случае записывается рядом с ромбом, соответствующим ассоциации.

Например, ассоциация под названием *Семья* может связывать следующие классы: *Муж, Жена, Ребенок,* как показано на рис. 2.10. В этом примере муж и жена должны присутствовать с семье обязательно (значение кратности у

соответствующего полюса ассоциации 1), а детей может быть произвольное количество или не быть вовсе (значение кратности у соответствующего полюса ассоциации 0..*).

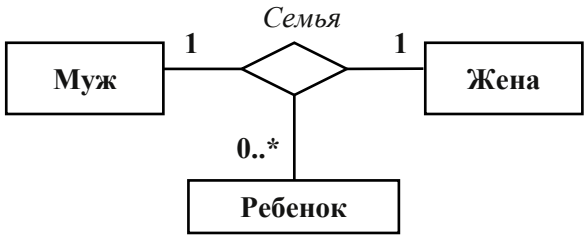


Рис. 2. 10. Графическое изображение n-арной ассоциации

Следует отметить, что *n*-арных ассоциаций лучше избегать, разбивая их на бинарные, так как большинство языков программирования, в том числе и C++, потребуют превращения таких ассоциаций в классы. Поэтому в языке UML предусмотрена возможность, которая позволяет объединять ассоциации между классами в классы ассоциаций или преобразовывать их в ассоциацию-класс.

Ассоциация-класс (association class) – это элемент модели, который имеет свойства, как ассоциации, так и класса, и предназначен для спецификации дополнительных свойств ассоциации в форме атрибутов и, возможно, операций класса.

Основанием для введения классов ассоциаций послужила возможность создания ассоциаций типа «многие-ко-многим». Атрибуты таких ассоциаций являются принадлежностью связей и не могут быть приписаны ни к одному из объектов-участников. Ассоциация-класс не только соединяет множество классов, но и определяет множество характеристик, которые принадлежат самому отношению и не принадлежат ни к одному из классов.

Графически ассоциация-класс изображается в форме символа класса и присоединяется к пути ассоциации пунктирной линией. Все элементы спецификации и графического представления ассоциации-класса аналогичны описанию элементов обычного класса, которое было рассмотрено выше.

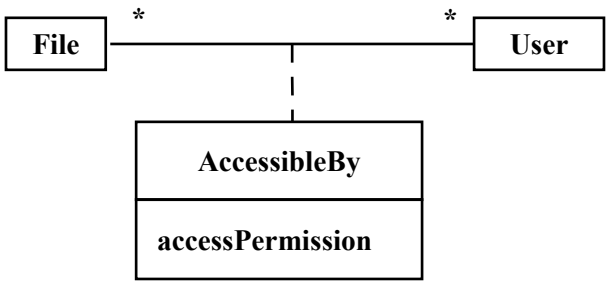


Рис. 2.11. Графическое изображение ассоциации-класса

В приведенном на рис. 2.11 примере атрибут *accessPermission (разрешениеДоступа)* относится и к файлу и к пользователю одновременно и не может быть прикреплен ни к одному из них без потери информации, поэтому введена ассоциация-класс *AccessibleBy (Доступно)*.

Итак, отношение ассоциации – это взаимосвязь между классами, по которой можно между ними перемещаться. После того, как классы связаны ассоциацией, они могут передавать друг другу сообщения, которые так же отражаются на диаграмме последовательности и кооперативной диаграмме.

Ассоциация является наиболее общей формой представления отношений в языке UML и служит для представления отношений между равноправными классами. Все другие типы отношений можно считать частным случаем отношения ассоциации. Однако, свойства этих типов отношений важны для построения диаграмм классов, так как они определяют дополнительные специфические характеристики взаимодействия между элементами классов. Данные типы отношений имеют специальные обозначения и относятся к базовым понятиям языка UML.

2.2.3. Отношение обобщения

Обобщение используется для моделирования одного из базовых принципов методологии объектно-ориентированного анализа и проектирования – наследования, благодаря которому можно создавать новые классы на основе уже существующих.

Обобщение (generalization) - это отношение между классом (суперклассом) и одной или несколькими его вариациями (подклассами). Суперкласс характеризуется общими атрибутами, операциями и ассоциациями. Подкласс добавляет к ним свои собственные атрибуты, операции и ассоциации.

Иначе говоря, подкласс наследует составляющие суперкласса, то есть класс-потомок обладает всеми свойствами и поведением класса предка, а также может иметь дополнительные свойства и поведение, которые отсутствуют у класса предка, то есть помимо наследуемых атрибутов и методов, каждый подкласс может иметь свои собственные уникальные атрибуты и операции.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения.

Графически обобщение обозначается не закрашенной стрелкой, которая указывает на суперкласс (класс-предок).

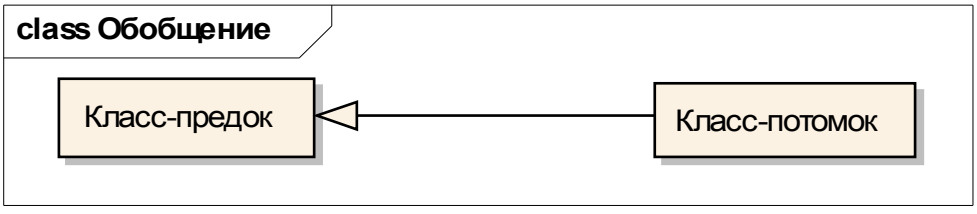


Рис. 2.12. Графическое изображение отношения обобщения

От одного класса-предка одновременно могут наследовать несколько классов-потомков. В этом случае на диаграмме классов для подобного отношения указывается несколько линий со стрелками, направленных от подкласса к классу-предку. Подклассы можно изображать с любой стороны от суперкласса, но обычно изображение соответствует по форме иерархическому дереву.

Важно отметить, что иерархия обобщения не должна содержать циклов.

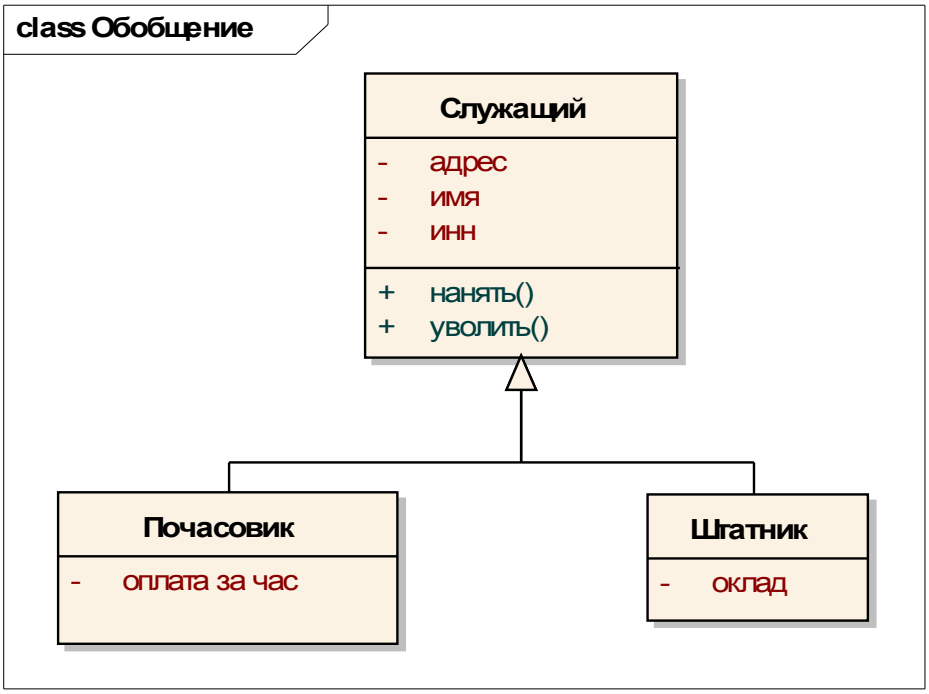


Рис. 2.13. Графическое изображение обобщения для нескольких классов-потомков

В приведенном на рис. 2.13 примере описаны два типа сотрудников: почасовики и штатные сотрудники. Оба этих класса наследуют от класса *Служащий*, который в данном случае является суперклассом. Общие для подкласса и суперкласса элементы размещаются в классе *Служащий*. Таким образом, классы *Почасовик* и *Штатник* наследуют от него атрибуты: *имя, адрес, инн* и операции: *нанять и уволить*.

Обобщение служит трем основным целям.

Первая цель – обеспечение поддержки полиморфизма, то есть операция может быть вызвана на уровне суперкласса, а компилятор объектно-ориентированного языка автоматически разрешит вызов метода, соответствующего классу вызывающего объекта. Полиморфизм увеличивает гибкость программного обеспечения, так как при добавлении нового класса этот класс автоматически наследует поведение суперкласса. Более того, новый подкласс не нарушает работу существующего кода.

Вторая цель обобщения состоит в структурировании описаний объектов, то есть при использовании обобщения, объекты упорядочиваются на основании их сходств и различий.

Третья цель состоит в обеспечении повторного использования кода, тем самым обобщение позволяет экономить усилия разработчиков, как при моделировании, так и при дальнейшей поддержке программы. Например, не требуется писать и поддерживать две различные копии операции *нанять* для каждого подкласса. Достаточно реализовать эту операцию в суперклассе и любые, сделанные в ней, изменения будут автоматически наследоваться подклассами.

2.2.4. Абстрактные классы

Рассмотренные выше обозначения относятся к ***конкретным классам***, то есть к классам, на основе которых могут быть созданы экземпляры или объекты. От них следует отличать *абстрактные классы*.

Абстрактный класс (abstract class) – это класс, который не имеет непосредственных экземпляров или объектов. При этом непосредственные экземпляры могут быть у потомков абстрактного класса, а конкретные классы могут иметь абстрактные подклассы, но у них, в свою очередь, обязательно должны быть конкретные потомки, так как листьями дерева наследования могут быть только конкретные классы.

Для обозначения абстрактного класса используется наклонный шрифт (курсив). Так же в языке UML принято соглашение о том, что любой текст, относящийся к абстрактному элементу модели, записывается курсивом.

Важно отметить, что в данном случае курсив при записи имени класса, в отличие от толщины шрифта, имеет принципиальное значение, поскольку является семантическим аспектом описания абстрактных элементов языка UML. По этой причине в процессе разработки модели программной системы следует внимательно записывать имена классов.

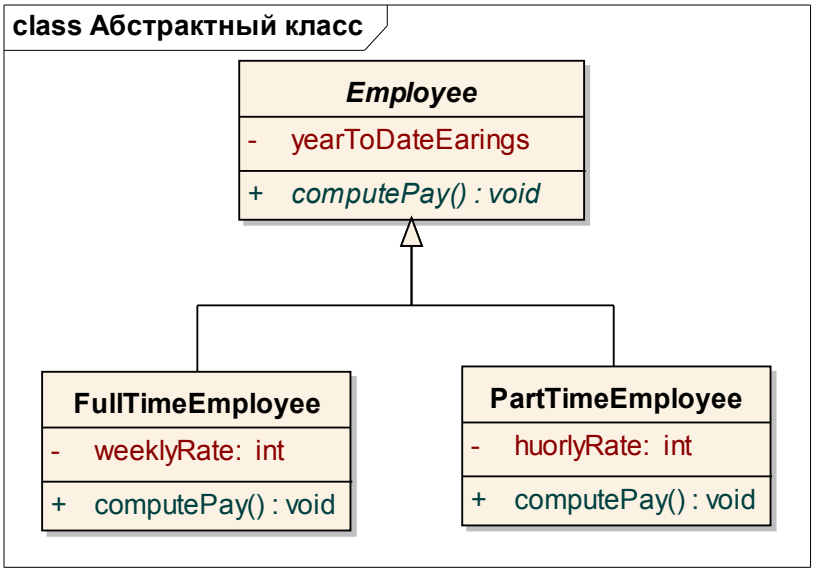


Рис. 2.14. Изображение абстрактного класса и абстрактной операции

Примером абстрактного класса (рис. 2.14) является класс *Employee* (Служащий). Все служащие имеют либо полную, либо частичную занятость. Класс *FullTimeEmployee* (служащий на полной занятости) и класс *PartTimeEmloyee* (служащий на частичной занятости) – это конкретные классы, так как они могут иметь непосредственные экземпляры.

Абстрактные классы могут использоваться для определения методов, которые будут наследоваться подклассами. В этом случае абстрактный класс может определять только сигнатуру (форму операции или процедуры) операции без реализующего ее метода. Такая операция называется *абстрактной*. Реализацию такой операции должен предоставить конкретный подкласс данного абстрактного класса. Конкретный подкласс не может содержать абстрактные операции, потому что в этом случае его объекты или экземпляры будут иметь неопределенные методы.

На рис. 2.14 показана абстрактная операция *computePay* (вычислить зарплату) класса *Employee* (Служащий). В этом классе определена только сигнатура операции, а каждый из конкретных подклассов должен предоставить метод, реализующий данную операцию.

Как уже говорилось, объектно-ориентированные языки, например C++, позволяют реализовать обобщение через наследование. Механизм наследования не только делает определение подклассов более удобным, но и позволяет более абстрактным образом работать с объектами. При этом наследование делает возможным полиморфизм, благодаря которому тип потомка может выступать в качестве представителя типа предка.

Особенностью языка C++ в данном случае является то, что подклассы в C++ не имеют общего предка, то есть иерархия классов может начинаться с произвольного класса. Класс может смешивать методы, разрешение которых осуществляется автоматически во время выполнения программы, со всеми прочими методами. При этом полиморфизм в C++ не является автоматическим, а управляется программистом при помощи квалификатора virtual. То есть, чтобы разрешить полиморфизм, метод нужно объявить как virtual. Все прочие методы будут вызываться в соответствии с типом ссылки на объект, а не в соответствии с фактическим типом порожденного класса, даже если в порожденном классе метод перекрывается.

```
class Hello {
   public:
         void method1 ()  {  cout << “hello\n”; }
         virtual void method2 () { cout<<”hello\n”; }
};

class Goodbye : public Hello {
   public:
      void method1 () {  cout << “goodbye\n”; }
      void method2 () {  cout << “goodbye\n”; }
};
```

```
int main ()  {
   Goodbye g;
   Hello& h = g;                          //  тот же объект, но ссылка базового типа
   g.method1();
   g.method2();
   h.method1();
   h.method2();
}
```

В результате не экран будет выведен следующий текст:

```
goodbye                 // method1 не виртуальный, вызывается из подкласса
goodbye                 // method2 виртуальный, вызывается из  подкласса
hello                   // method1 не виртуальный, вызывается из базового класса
goodbye                 // method2 виртуальный, вызывается из  базового класса
```

C++ не позволяет запретить перекрытие методов. Однако в иерархии, где часть методов перекрывается, а часть – нет, отсутствие модификатора virtual может указывать на желание автора сохранить базовый метод в неизменности.

Спецификация доступа в C++ включена в синтаксис описания наследования. Порожденный класс может иметь тип доступа public, protected или private.

Открытое наследование подразумевает, что все открытые методы базового класса останутся открытыми и в порожденном классе.

При закрытом наследовании все методы базового класса становятся закрытыми. Такое наследование подразумевает, что базовый класс используется для упрощения реализации порожденного класса, но при этом порожденный класс не является частным случаем базового класса.

Защищенное наследование подразумевает, что открытые методы базового класса доступны только потомкам данного класса. На практике это используется довольно редко.

Абстрактные классы получаются при включении, по крайней мере, одного чисто виртуального метода, при объявлении которого используется синтаксис инициализации нулем: void fn() = 0. Такие классы не могут иметь экземпляров, а порожденные от них классы остаются абстрактными до тех пор, пока не реализуют все унаследованные чисто виртуальные методы.

2.2.5. Множественное наследование

Множественное наследование позволяет классу иметь несколько суперклассов и наследовать составляющие от всех предков. Это позволяет смешивать информацию из нескольких источников. Множественное наследование является более сложной формой обобщения по сравнению с единичным наследованием, ограничивающим иерархию классов до дерева.

Преимущество множественного наследования заключается в увеличении возможностей спецификации классов и их повторного использования. Недостаток – усложнение концепций и реализации.

Чаще всего множественное наследование используется для наследования от множества несовместных классов. Каждый подкласс является потомком одного класса из данного множества.

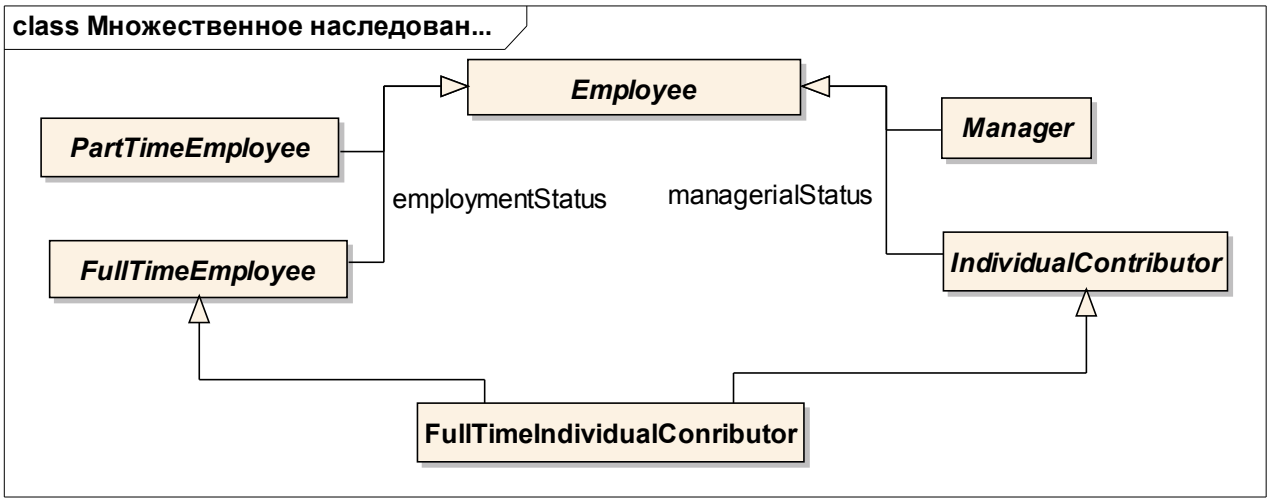


Рис. 2.15. Множественное наследование от несовместных классов

На рис. 2.15 классы *FullTimeEmployee* и *PartTimeEmployee* являются несовместными, то есть каждый сотрудник может принадлежать ровно одному из них. Классы *Manager* и *IndividualContributor* тоже являются несовместными. Класс-потомок *FullTimeIndividualContributor* сочетает в себе составляющие классов *FullTimeEmployee* и *IndividualContributor*.

Каждое обобщение должно производиться только по одному аспекту. Множественное наследование можно использовать только в том случае, если класс нельзя уточнить по нескольким независимым аспектам.

В приведенном на рис. 2.15 примере класс *Employee* уточняется по занятости и участию в управлении, поэтому модель содержит два набора обобщений. При этом подкласс наследует составляющие от каждого суперкласса только один раз, даже если к нему идут несколько путей по графу обобщений. То есть *FullTimeIndividualContributor* наследует составляющие класса *Employee* по двум путям: через employmentStatus и managerialStatus. Однако каждый *FullTimeIndividualContributor* будет иметь только одну копию составляющих класса *Employee*.

Язык C++ поддерживает множественное наследование: каждый класс может иметь один или несколько суперклассов. На практике оптимальная комбинация родительских классов представляет собой смесь конкретных типов и абстрактных классов, описывающих требования к поведению, благодаря этому можно определять основные реализации методов там, где это удобно.

```
class Account {                                                    // не абстрактный класс
   private:
      float balance;
   public:

        /* предположим, что реализация метода Post может быть разной для разных
типов счетов. Базовая реализация – это вариант, предполагаемый по умолчанию. */

      virtual void Post ( float amount ) { . . . }
         float Balance () { return balance; }
};

class InterestBearingAcct {                                        // абстрактный класс
   private:
         float rate;
   public:
         virtual float CalcInterest () = 0;                        // чисто виртуальный метод
         float Rate () { return rate;}
};

class SavingsAccount : public Account, public InterestBearingAcct {
    public:
          virtual float CalcInterest () {
           . . . . . .                                             //расчет причитающихся процентов
          }
};
```

2.2.6. Отношение агрегации

Следующий частный случай отношения ассоциации, который описывает объекты, состоящие из частей, и применяется для представления системных взаимосвязей типа «часть-целое» – это отношение агрегации.

Агрегация (aggregation) – это направленное соотношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности. Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами.

Раскрывая внутреннюю структуру системы, агрегация показывает - из каких элементов состоит система, и как они связаны между собой. Другими словами, данное отношение описывает разбиение сложной системы на более простые составные части.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Отношение агрегации предполагает, что части, отделенные от целого, могут продолжать своё существование независимо от него.

Рассматриваемое в таком аспекте деление системы на составные части также представляет собой некоторую иерархию. Однако данная иерархия принципиально отличается от той, которая порождается отношением обобщения, то есть агрегирование принципиально отличается от наследования. Отличие заключается в том, что части системы имеют другой тип сущности, а значит, не наследуют ее свойства и поведение, поскольку являются вполне самостоятельными сущностями. Более того, части целого обладают своими собственными атрибутами и операциями, которые отличаются от атрибутов и операций целого.

Другими словами, агрегирование никак не затрагивает сам агрегат, то есть у него могут быть свои собственные атрибуты и методы, и в случае агрегирования части не «растворяются» в целом, а остаются отдельными частями в его составе.

Важно отметить, что агрегациями могут служить только бинарные ассоциации.

Графически отношение агрегации изображается сплошной линией с ромбом, который ставится около полюса являющегося агрегатом (класс-контейнер).

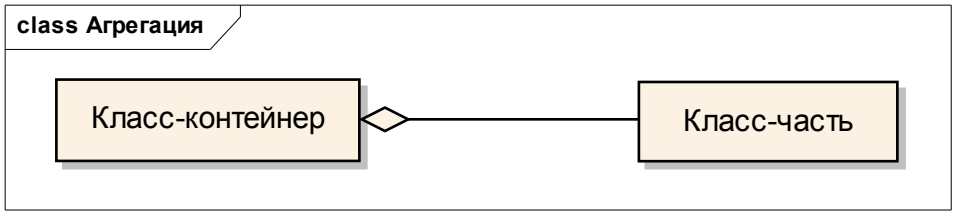


Рис. 2.16. Графическое изображение отношения агрегации

В качестве примера отношения агрегации можно рассмотреть всем известное разделение персонального компьютера на составные части. Используя обозначения языка UML, компонентный состав персонального компьютера можно представить в виде соответствующей диаграммы классов, которая в данном случае иллюстрирует отношение агрегации.

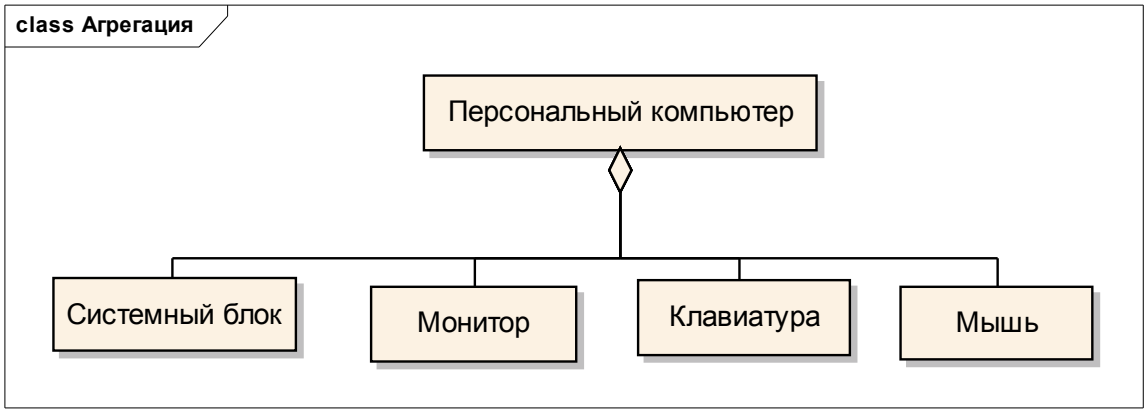


Рис. 2.17. Пример отношения агрегации.

Решение об использовании агрегации является субъективным и может быть достаточно произвольным. В общем случае, если два объекта связаны отношением «часть-целое», их следует моделировать при помощи агрегации. Если же объекты рассматриваются как независимые, хотя между ними и может возникать связь, это будет ассоциация.

2.2.7. Отношение композиции

Отношение композиции - это частный случай отношения агрегации, который является более сильной формой отношения «часть-целое».

Композиция (composition) – это отношение, при котором части принадлежат целому, причем часть может принадлежать не более чем одному композиту. Более того, эта составляющая часть автоматически получает срок жизни, совпадающий со сроком жизни целого. Поэтому если класс-композит (целое) удаляется, все его части удаляются вместе с ним. Это удобно для программирования, так как удаление объекта автоматически вызывает удаление всех его составляющих, если он образует их композицию.

Разница между агрегацией и композицией состоит в том, что при агрегации объекты-части могут существовать сами по себе, а при композиции – нет.

Графически отношение композиции изображается сплошной линией с закрашенным ромбом на конце, который указывает на класс-композит (класс-целое).

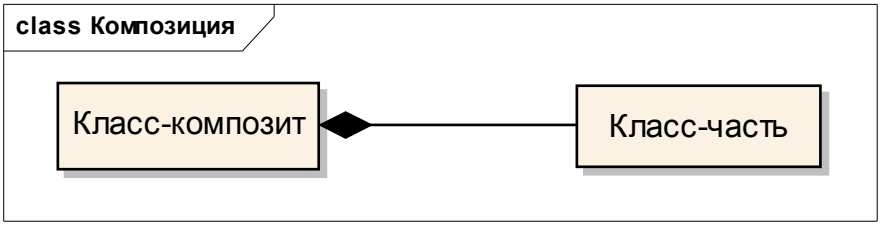


Рис. 2.18. Графическое изображение отношения композиции

Для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, могут указываться кратности отдельных полюсов, которые в общем случае не обязательны.

Важно отметить, что кратность агрегированного полюса ассоциации не должна превышать 1, так как часть не может принадлежать более чем одному композиту.

Практическим примером отношения композиции может служить окно графического интерфейса программы, которое может состоять из заголовка, полос прокрутки, рабочей области, строки состояния и главного меню. Окно

программы является классом-композитом, а его составные элементы также являются отдельными классами.

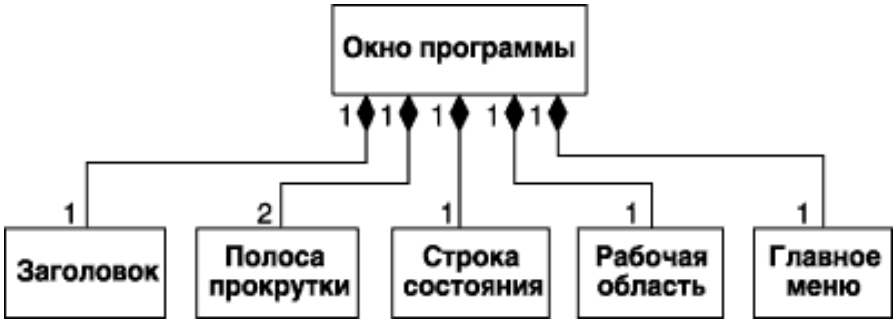


Рис. 2.19. Пример отношения композиции

Агрегация и композиция в языке C++ реализуются по тем же правилам, что и обычные ассоциации, но при этом имеются некоторые особенности, связанные с владением и принадлежностью объектов, связанных отношением агрегации или композиции. Рассмотрим, чем же отличается реализация агрегации от реализации композиции.

```
class Heater  {
. . . . . .
};

class TemperatureController {
    private:
   Heater h;
. . . . . .
public:
   TemperatureController(Location);
   ~TemratureController();
. . . . . .

};
```

В этом примере класс TemperatureController это, несомненно, целое, а экземпляр класса Heater - одна из его частей.

В случае класса TemperatureController используется агрегация по значению. Эта разновидность физического включения означает, что объект класса Heater не существует отдельно от объемлющего экземпляра класса TemperatureController. Другими словами, в представленном примере реализована композиция классов Heater и TemperatureController.

Менее обязывающим является включение по указателю, то есть агрегацию по значению или композицию можно изменить следующим образом:

```
Heater* h;
```

В этом случае класс TemperatureController по-прежнему означает целое, но его часть, экземпляр класса Heater, содержится в целом косвенно. Теперь эти объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо.

Можно также описать h как ссылку на Heater :

Heater& h;

Однако следует помнить, что в этом случае семантика инициализации и модификации этого объекта будет совершенно отличной от семантики указателей. Это подробно обсуждалось в реализации ассоциаций.

Агрегация является направленным отношением, как и всякое отношение "целое/часть". То есть объект Heater входит в объект TemperatureController, а не наоборот. При этом следует отметить, что физическое вхождение одного в другое нельзя "зациклить", а вот указатели - можно (каждый из двух объектов может содержать указатель на другой).

Рассмотрим еще один пример, иллюстрирующий реализацию агрегации и композиции и связанную с данными отношениями проблему владения или принадлежности объектов.

В абстрактном огороде одновременно растет много растений, и от удаления или замены одного из них огород не становится другим огородом. Если уничтожается огород, растения остаются (их ведь можно пересадить). Другими словами, огород и растения имеют свои отдельные и независимые сроки жизни. Это достигается благодаря тому, что огород содержит не сами объекты Plant, а указатели на них. Напротив, план выращивания физически содержится в каждом экземпляре огорода и погибает вместе с ним, поэтому объект GrowingPlan внутренне связан с объектом Garden и не существует независимо.

```
class Plant {
. . . . . .
};

class GrowingPlan {
. . . . . .
};
class Garden {
protected:
   Plant* repPlants[100];
   GrowingPlan repPlan;
. . . . . .
public:
   Garden ();
   virtual ~Garden ();
. . . . . .
};
```

Конечно, как уже говорилось, 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 совершенно различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. "Лакмусовая бумажка" для выявления агрегации такова: если (и только если) налицо отношение "целое/часть" между объектами, их классы должны находиться в отношении агрегации друг с другом.

Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело - с наследованием или агрегацией - будьте осторожны. Если вы не уверены, что налицо отношение общего и частного, вместо наследования лучше применить агрегацию или что-нибудь еще.

Итак, мы рассмотрели основные элементы нотации языка UML, которые применяются при проектировании диаграмм классов и особенности их реализации. Важно отметить, что этот тип диаграмм является основным при моделировании объектно-ориентированной системы, так как позволяет наглядно изобразить структуру классов разрабатываемого приложения.

2.3. Пакеты.

Модель классов можно уместить на одной странице, если вы решаете задачу небольшого или среднего размера, но большую задачу бывает трудно охватить целиком, поэтому требуется механизм пакетов, позволяющий разместить различные конструкции по определенным группам в соответствии с описанием задачи.

Пакеты (package) применяют для того, чтобы разбивать большие модели на части, группируя классы, обладающие некоторой общностью. Это в значительной мере упрощает понимание и поддержку проектируемой модели.

Существует несколько наиболее распространенных подходов к группировке.

Группировка по стереотипу. Стереотипы – это механизм, позволяющий разделять классы на категории. В UML определены три основных стереотипа:

- граница;
- объект;
- управление.

Пограничные классы (boundary classes) расположены на границе проектируемой системы. Они включают все формы взаимодействия с

окружающей средой: отчеты, формы документов, интерфейсы с аппаратурой, пользователями и другими системами.

Классы-сущности (entity classes) содержат информацию о классах внутреннего устройства системы, т.е. являющиеся сущностью данной системы.

Управляющие классы (control classes) отвечают за координацию действий других классов, за их взаимодействие между собой.

Группировка по функциональности. Помимо упомянутых выше стереотипов можно создавать и свои собственные стереотипы (темы пакетов), то есть тема пакета может быть достаточно произвольной. Темой может быть основной класс, основные отношения, важнейшие аспекты функциональности. Например, модель классов компилятора можно было бы разделить на пакеты лексического анализа, разбора, семантического анализа, генерации кода и оптимизации.

Графически пакет изображается прямоугольником с закладкой. Закладка подразумевает вложенность содержимого и должна вызывать ассоциации с папкой.

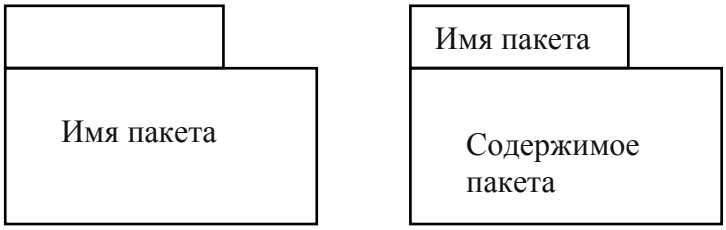


Рис. 2.20. Графическое изображение пакетов в языке UML

Пакет может быть изображен без своих членов. В этом случае имя пакета можно указать в большом прямоугольнике. Имя пакета должно быть уникальным в пределах рассматриваемой модели. Если изображаются классы-члены пакета, то тип пакета пишется внутри символа пакета (на закладке), а внутри большого прямоугольника перечисляются классы-члены пакета при помощи символа класса.

В UML для установления взаимодействий между пакетами в основном используется *отношение зависимости*.

Зависимость (dependency) – это однонаправленное отношение, показывающее, что класс зависит от определений, сделанных в другом классе. Зависимость означает отношение типа «поставщик-клиент», когда модификация поставщика может оказать влияние на одного или нескольких клиентов. Клиент – элемент модели, зависимый в некотором контексте от элемента или элементов поставщика. То есть семантика клиента не является полной без поставщика.

Зависимость между двумя пакетами существует в том случае, если между двумя любыми классами из этих пакетов существует любая взаимосвязь.

Отношение зависимости изображают в виде пунктирной линии со стрелкой, направленной от зависимого класса к классу-поставщику.

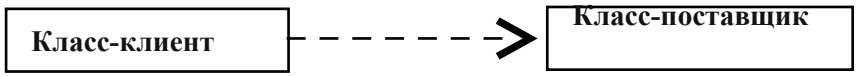


Рис. 2. 21. Графическое изображение отношения зависимости

Таким образом, диаграмма пакетов представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости – это лишь одна из форм диаграммы классов.

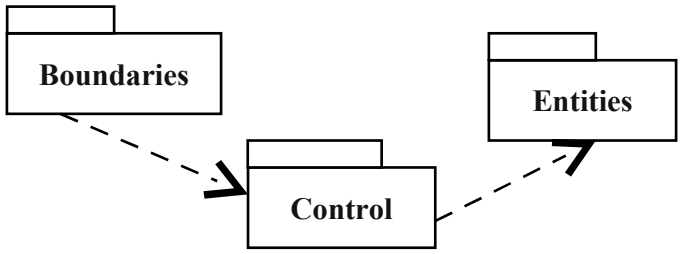


Рис. 2.22. Диаграмма пакетов

Итак, классы – это строительные блоки любой объектно-ориентированной системы. Они представляют собой описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой.

При проектировании любых объектно-ориентированных систем диаграммы классов обязательны.

Информация с диаграммы классов напрямую отображается в исходный код приложения. Для этого в большинстве существующих инструментов UML-моделирования предусмотрена возможность генерации кода для определенного языка программирования.

Таким образом, диаграмма классов – это отправная точка процесса разработки объектно-ориентированных программных систем.

После изучения статической структуры проектируемой системы, которая описывается с помощью модели классов, можно уделить внимание изменениям объектов и их отношений с течением времени, то есть перейти к разработке модели состояний.

Упражнения.

1. Создайте модель классов для описания неориентированных графов. Неориентированный граф состоит из множества вершин и множества дуг. Дуга соединяет между собой пару вершин. Модель должна описывать только структуру графа (соединения между вершинами) и не должна включать сведения о размещении дуг и вершин.
2. Расширьте диаграмму классов из предыдущей задачи, добавив в нее сведения о расположении вершин графа, толщине и цвете дуг. Добавьте также названия вершин и дуг.
3. Создайте модель классов для описания ориентированных графов. Ориентированный граф отличается от неориентированного тем, что его дуги имеют направления.
4. Определите перечисленные ниже отношения как обобщения, агрегации или ассоциации. Ответ поясните.
    1) У страны есть столица.
    2) Файл – это обычный файл или файл каталога.
    3) Файлы содержат записи.
    4) Человек использует язык программирования для выполнения проекта.
    5) Модемы и клавиатуры являются устройствами ввода-вывода.
    6) Классы могут иметь несколько атрибутов.
    7) Маршрут соединяет два города.
    8) Студент слушает курс лекций профессора.
5. Создайте диаграмму классов для компьютерной системы верстки газет. Система предназначена для работы с газетными страницами, на которых могут располагаться колонки текста. Пользователь может изменять ширину и длину колонки, перемещать ее по странице или переносить с одной страницы на другую. Колонка может размещаться одновременно на нескольких страницах. Если пользователь изменяет текст на одной странице, изменения должны появляться автоматически и на других страницах.
6. Подготовьте диаграмму классов для задачи об обедающих философах. Пять философов сидят за круглым столом, на котором лежат пять вилок. Каждый философ может дотянуться до двух вилок (по одной с каждой стороны). Каждая вилка может быть взята одним из двух философов. Вилка может лежать либо на столе, либо находиться в руке у философа. Философ может взять только две вилки.
7. Разработайте диаграмму классов для подписки на журналы. Один человек может быть подписан на несколько журналов. На один журнал может быть подписано несколько человек. Для каждой подписки необходимо отслеживать дату и размер каждого платежа, а также текущий срок окончания подписки.

8. Создайте диаграмму классов для редактора графических документов, поддерживающего группировку объектов. Пусть документ состоит из нескольких листов. На каждом листе могут располагаться объекты рисунка, включая текст, геометрические объекты и группы. Группа – это множество объектов рисунка, в которое могут входить другие группы. Группа должна содержать, по меньшей мере, два объекта. Объект может быть непосредственным членом только одной группы. К геометрическим объектам относятся окружности, эллипсы, прямоугольники, отрезки и квадраты.
9. Разработайте диаграмму классов простого редактора диаграмм. Лист диаграммы – это совокупность линий и прямоугольников. Линия – это последовательность прямолинейных сегментов, соединяющих два прямоугольника. Каждый сегмент определяется двумя конечными точками. Точка может принадлежать вертикальному и горизонтальному сегменту одной и той же линии. Для последующего редактирования выделение – это совокупность линий и прямоугольников, выделенная пользователем. Также следует предусмотреть возможность использования буфера. Буфер – это совокупность линий и прямоугольников, вырезанная или скопированная с листа.
10. Разработайте диаграмму классов для системы продажи товаров в интернет-магазине. Клиенты магазина имеют доступ к каталогу товаров, поддержку которого осуществляет интернет-магазин. В каталоге все товары распределены по разделам. Менеджер магазина может добавлять товары в каталог и удалять их. О каждом товаре доступна следующая информация: название, количество, цена, дата выпуска. При отборе клиентами товаров поддерживается виртуальная корзина. Любое наименование может быть добавлено или изъято из корзины. Корзина может быть очищена. По окончании выбора товаров производится оформление заказа и регистрация покупателя. При оформлении заказа указываются: номер, стоимость, дата, форма оплаты. При регистрации клиент указывает имя, фамилию, адрес доставки.
11. Реализуйте на языке C++:
    1) ассоциацию типа «один к одному», прослеживаемую в обоих направлениях;
    2) ассоциацию типа «один ко многим», прослеживаемую в направлении от одного ко многим;
    3) ассоциацию типа «многие ко многим», прослеживаемую в обоих направлениях.
12. Реализуйте на языке C++ модель классов, полученную в упражнении 7.

3. Моделирование состояний

Модель состояний описывает последовательности операций, происходящих в системе в ответ на внешнее воздействие, иначе говоря, она показывает, как элементы системы и система в целом переходят из одного состояния в другое. Модель состояний может включать несколько диаграмм состояний, по одной на каждый класс, поведение которого во времени важно для проектируемого приложения. Класс, имеющий несколько состояний, обладает важным поведением во времени. Если класс обладает только одним состоянием, его поведение во времени можно игнорировать.

3.1. Диаграммы состояний

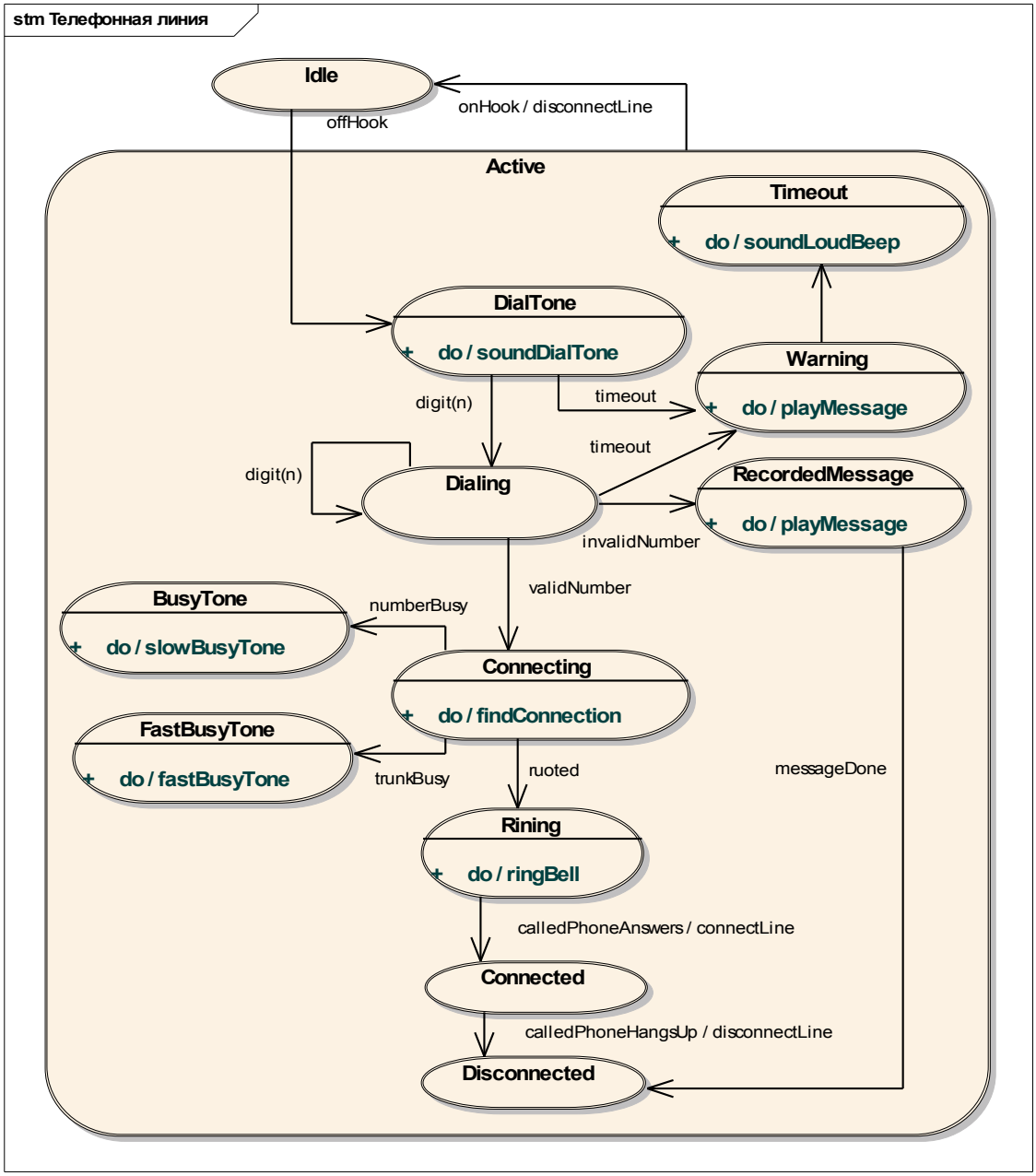


Рис. 3.1. Диаграмма состояний телефонной линии

Диаграммы состояний являются средством для описания поведения систем, другими словами, диаграмма состояний описывает последовательности состояний, вызываемых последовательностями событий.

Диаграмма состояний (state machine или statechart diagramm) – это стандартная концепция информатики (графическое представление конечного автомата), связывающая события и состояния.

Иначе говоря, диаграмма состояний – это ориентированный граф, вершинами или узлами которого являются состояния, а направленные дуги – переходы между ними.

С помощью диаграмм состояний специфицируются все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Для обозначения диаграммы состояний в графической нотации унифицированного языка моделирования используется прямоугольник. Название диаграммы указывается в пятиугольном теге в левом верхнем углу. Внутри прямоугольника изображаются состояния и переходы, образующие диаграмму состояний.

На рис. 6.1 приведен пример диаграммы состояний для телефонной линии.

Моделирование состояний основывается на понятиях события и состояния.

3.1.1. События и состояния

Событие (event) **–** происшествие, случившееся в определенный момент времени. Другими словами, *события* – это внешние воздействия. Например, нажатие кнопки или вылет рейса. События происходят в результате выполнения некоторого действия в системе или ее окружении, поэтому в описании задачи часто события соответствуют глаголам в прошедшем времени (питание включено) или выполнению некоторого условия (температура опустилась).

Состояние (state) – абстракция значений и связей объекта, то есть множества значений и связей группируются в состояние в соответствии с массовым поведением объектов. Состояние описывает отклик объекта на получаемые события.

В формулировке задач состояния часто соответствуют глаголам или деепричастиям (ожидает, дозванивается) или выполнению некоторого условия (включен, ниже точки замерзания).

Объекты класса обладают конечным числом возможных состояний. В конкретный момент времени каждый объект может находиться ровно в одном состоянии. В течение времени своего существования объекты могут проходить через одно или несколько состояний.

Между событиями и состояниями существует некоторая симметрия, которая показана на рис. 3.2. События – это точки на линии времени, а состояния – интервалы. Другими словами, событие соответствует интервалу между двумя точками, обозначающими два полученных объектом события.

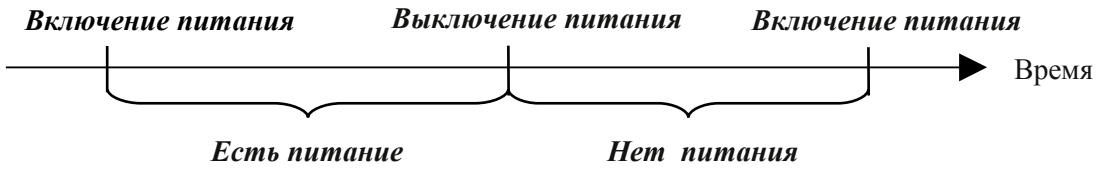


Рис. 3.2. События и состояния

В UML у состояния возможны следующие элементы спецификации:

- имя;
- деятельность при входе;
- текущая деятельность;
- деятельность при выходе;

Графически состояния обозначаются в виде прямоугольника с закругленными краями, в котором пишется имя состояния.

Имя состояния представляет собой законченное предложение, которое отражает содержательный смысл или семантику данного состояния. Имя состояния пишется полужирным шрифтом с большой буквы посередине прямоугольника. Имена состояний должны быть уникальными в рамках диаграммы.

Как исключение, имя у состояния может отсутствовать, то есть оно необязательно для некоторых состояний. В этом случае состояние является *анонимным*. Если на одной диаграмме несколько анонимных состояний, то все они должны различаться между собой.

Рис. 3.3. Графическое изображение состояний

Прямоугольник состояния может быть разделен на две части. В этом случае, в первой из них записывается имя состояния, а во второй – список

некоторых действий или переходов в данном состоянии, иначе говоря, деятельность в данном состоянии.

3.1.2. Деятельность

Деятельность (activity) - это поведение, реализуемое объектом, пока он находится в данном состоянии.

Как уже говорилось, деятельность может выполняться при входе в состояние и выходе из него, а так же при наступлении какого-либо иного события в состоянии.

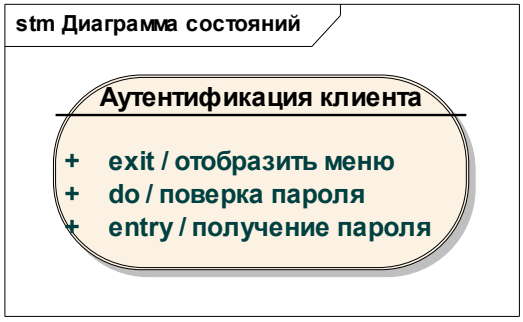


Рис. 3.4. Пример состояния с деятельностью

Текущая деятельность (do activity) – деятельность, занимающая некоторый промежуток времени. По определению такая деятельность может выполняться только в некотором состоянии и не может прикрепляться к переходу. Текущая деятельность включает непрерывные операции, такие как отображение картинки на экране, а так же последовательные операции, завершающиеся по прошествии некоторого промежутка времени (закрытие клапана).

Для обозначения текущей деятельности используется ключевое слово ***do*** и символ косой черты /, после которого описывается деятельность. Этот вид деятельности изображается внутри значка состояния.

Деятельность при входе изображается внутри значка состояния с ключевым словом ***entry*** и символом /. Такая деятельность выполняется при входе в состояние по любому из входящих переходов. Деятельность при входе эквивалентна прикреплению той же деятельности к каждому из входящих переходов. Если входящий переход имеет свою собственную деятельность, то она выполняется в первую очередь.

Деятельность при выходе используется достаточно редко, но иногда оказывается полезной. Деятельность при выходе указывается внутри символа состояния после ключевого слова ***exit*** и /. Эта деятельность выполняется в первую очередь при выходе из состояния по любому из переходов.

3.1.3. Переход

Переход (transition) – является направленным отношением между двумя состояниями, одно из которых является *вершиной источником (source vertex)*, а другое *целевой вершиной (target vertex)*.

Другими словами, переход – это смена одного состояния другим. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями, причем последовательность изменения состояний упорядочена во времени, то есть каждое последующее состояние может наступить только после предшествующего ему состояния. Таким образом поведение на диаграммах состояний моделируется как последовательное перемещение от состояния к состоянию по связывающим их дугам или переходам.

Говорят, что переход запускается или срабатывает при смене исходного состояния целевым. Исходное и целевое состояния обычно отличаются друг от друга, но иногда могут и совпадать. В этом случае объект переходит в тоже состояние, в котором он находился в первоначальный момент времени, то есть переход будет *рефлексивным.*

Переход запускается, когда происходит связанное с ним событие. Выбор целевого состояния зависит как от исходного состояния, так и от полученного события. Переход невозможно прервать, и если уж он запустился, то все действия, определенные в нем, должны отработать, прежде чем элемент системы сможет отреагировать на какое-либо следующее событие в системе. После окончания перехода элемент оказывается в новом (целевом) состоянии, то есть обработка текущего события считается завершенной и элемент может реагировать на следующие события.

На диаграмме все переходы обозначаются в виде стрелки, начинающейся на первоначальном состоянии и указывающей на последующее (целевое).

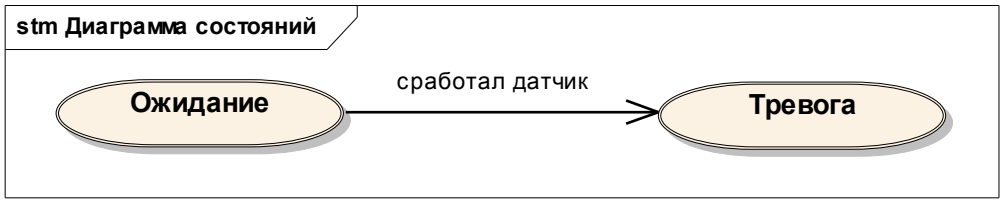


Рис. 3.5. Графическое изображение перехода

Событие, вызвавшее переход может быть указано в качестве метки перехода. В этом случае рядом со стрелкой перехода записывается строка текста, имеющая следующий формат:
<имя события> (список параметров) [сторожевое условие] /действие.

Имя события указывается с маленькой буквы и может быть выделено курсивом. Обычно в качестве имен переходов задают имена операций, вызываемых у тех или иных элементов системы. После имени такого события следуют круглые скобки для задания параметров соответствующей операции. Если таких параметров нет, то список может отсутствовать.

Если с переходом не связано какое-либо событие, то он срабатывает после завершения деятельности в начальном состоянии. В этом случае рядом с символом перехода имя или название события не указывается.

Как уже говорилось, переход вызывается связанным с ним событием, но переход может и не произойти, если для него есть сторожевое условие, которое приводит к игнорированию события.

Сторожевое условие или ограничение (guard condition) - это логическое выражение, которое должно быть истинным, чтобы переход сработал. Иначе говоря, сторожевое условие определяет, когда переход может, а когда не может осуществиться.

Переход со сторожевым условием запускается в тот момент, когда осуществляется соответствующее событие, но только если в этот же момент выполнено его сторожевое условие. Оно проверяется только один раз, в тот момент, когда осуществляется событие, и если условие выполняется – происходит переход. Другими словами, сторожевое условие может принимать либо значение «истина», и соответствующий переход срабатывает, либо значение «ложь», тогда переход не срабатывает, даже если произошло событие, инициирующее данный переход. Сторожевое условие (ограничение) записывается в квадратных скобках после метки события и должно представлять из себя логическое выражение, возвращающее булевское значение.

При моделировании состояний может оказаться, что из одного и того же состояния выходят несколько переходов. Если эти переходы вызваны разными событиями, то никаких проблем не возникнет, но если переходы из одного и того же состояния не содержат метки события или вызваны идентичными событиями, то тогда произойдет конфликт переходов.

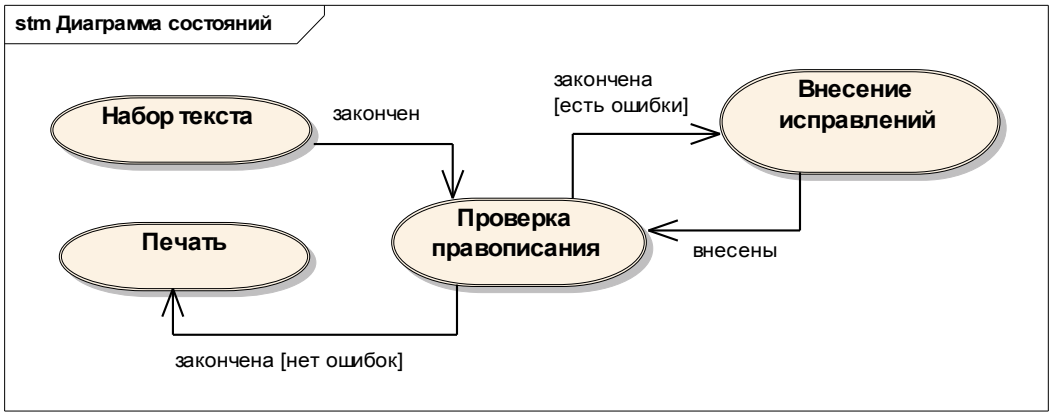


Рис. 3.6. Пример применения переходов со сторожевыми условиями

В примере на рис. 3.6 показана ситуация, когда из одного состояния («Проверка правописания») могут выходить несколько переходов вызванных одним и тем же событием («завершена»). В этом случае каждый такой переход должен содержать сторожевое условие и никакие два из них не должны одновременно принимать значение «истина». Иначе будет не понятно, по какому из переходов происходит дальнейшая смена состояний, и такая модель системы будет считаться некорректной.

Помимо имени и сторожевого условия около символа перехода может быть указана и деятельность при переходе. На диаграммах состояний деятельность при переходе изображается с помощью символа « / », после которого ставится название или описание деятельности. Деятельность указывается после вызывающего ее события около символа перехода.

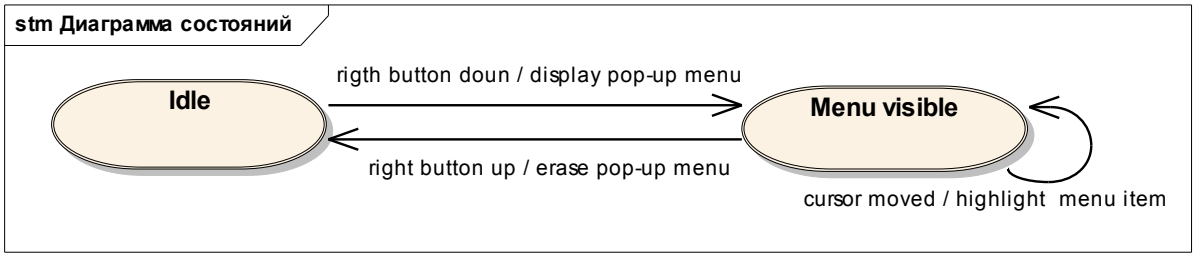


Рис. 3.7. Пример деятельности при переходе

На рис. 3.7 показан пример диаграммы состояний для всплывающего меню, иллюстрирующий применение деятельности при переходе. При нажатии правой кнопки мыши меню изображается на экране. Когда пользователь отпускает кнопку, меню исчезает. Пока меню отображается на экране, в нем подсвечивается один элемент, над которым в данный момент находится указатель курсора.

Если к состоянию привязаны несколько видов деятельности, включая деятельность при переходе и деятельность в состоянии, то они выполняются в следующем порядке:

- деятельность при входящем переходе;
- деятельность при входе;
- текущая деятельность;
- деятельность при выходе;
- деятельность при исходящем переходе.

3.1.4. Псевдосостояния

Кроме обычных состояний на диаграмме состояний могут размещаться *псевдосостояния*.

Псевдосостояния – это абстрактный элемент модели, который включает в себя различные типы вспомогательных вершин в графе конечного автомата,

то есть это такие вершины, которые имеют форму состояния, но не обладают поведением.

В UML определены несколько типов псевдосотояний.

Начальное состояние (initial) – разновидность псевдосостояния, обозначающее начало выполнения процесса изменения состояний конечного автомата. В этом состоянии объект находится сразу после создания.

На диаграмме состояний начальное состояние может быть только одно и может иметь только один выходящий переход, который называют начальным переходом, который может содержать метку события, но не может содержать сторожевого условия. Обозначается начальное состояние в виде закрашенного кружка.

Конечное состояние (final) – разновидность псевдосостояния, обозначающая прекращение процесса изменения состояний конечного автомата. В этом состоянии должен находиться моделируемый объект после завершения работы конечного автомата.

Вход в конечное состояние означает уничтожение объекта и обозначается закрашенным кругом внутри белого круга («бычий глаз»). Конечных состояний может быть несколько или не быть вообще.

На рис. 3.8 показан упрощенный вариант диаграммы состояний для игры в шахматы с начальным состоянием по умолчанию (черный кружок) и конечным состоянием по умолчанию («бычий глаз»).

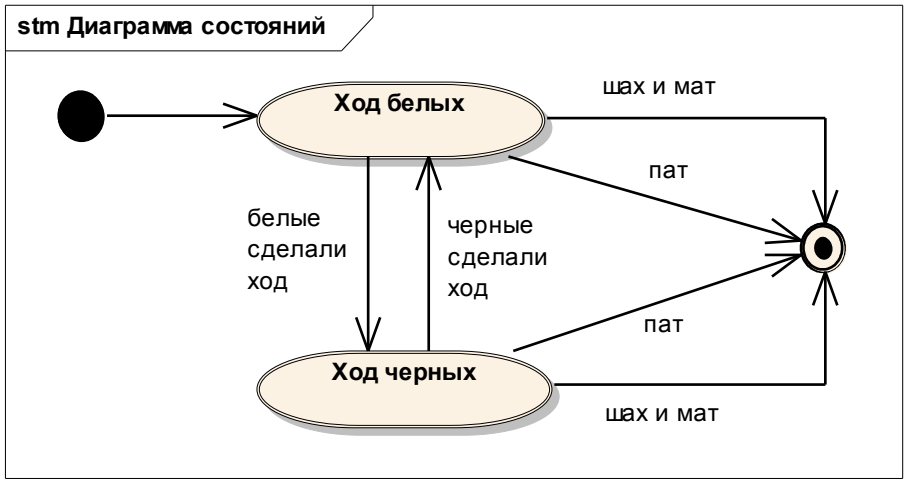


Рис. 3.8. Начальное и конечное состояния на диаграмме состояний

Такие диаграммы обычно называют одноразовыми, так как они описывают объекты с конечным сроком существования.

Начальное и конечное состояния можно обозначать точками входа и выхода.

Точка входа (entry point) – используется для соединения некоторого внешнего перехода оканчивающегося в этой точке с некоторым внутренним переходом, исходящим из этой точки.

Точка входа изображается окружность на границе диаграммы состояний и может иметь необязательное имя, которое пишется с большой буквы около символа входа.

Точка выхода (exit point) – используется для соединения внутреннего перехода, оканчивающегося в этой точке с некоторым внешним переходом, исходящим из этой точки выхода.

Точка выхода изображается в виде окружности с крестом на границе диаграммы состояний и может иметь необязательное имя, которое пишется с большой буквы рядом с символом выхода.

На рис. 3.9 показан предыдущий пример диаграммы состояний для игры в шахматы с использованием точки входа и точек выхода.

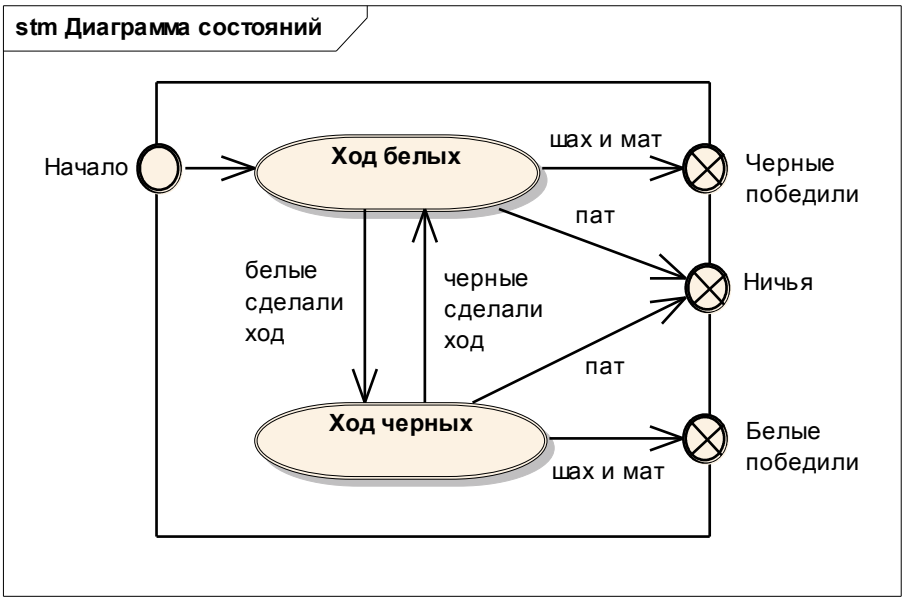


Рис. 3.9. Диаграмма состояний с точками входа и выхода

3.1.5. Составные состояния и подсостояния.

Моделирование сложных объектно-ориентированных систем, как правило, связано с многоуровневым представлением их состояний. В этом случае возникает необходимость детализировать отдельные состояния, сделав их составными, то есть для представления модели состояний применяются диаграммы высокого уровня, отдельные детали которых раскрываются на поддиаграммах.

Этот принцип аналогичен подстановке макросов в языках программирования.

Составное состояние (composite state) – сложное состояние, которое состоит из других вложенных в него состояний. Составное состояние называют также состоянием-композитом. Вложенные состояния выступают по отношению к составному как ***подсостояния (substate).***

Графически все элементы диаграммы, которые соответствуют вложенным состояниям, изображаются внутри символа составного состояния. В этом случае все размеры графического символа составного состояния увеличиваются таким образом, чтобы вместить в себя все подсостояния.

Совокупность вложенных последовательных подсосотояний представляет собой вложенный конечный автомат.

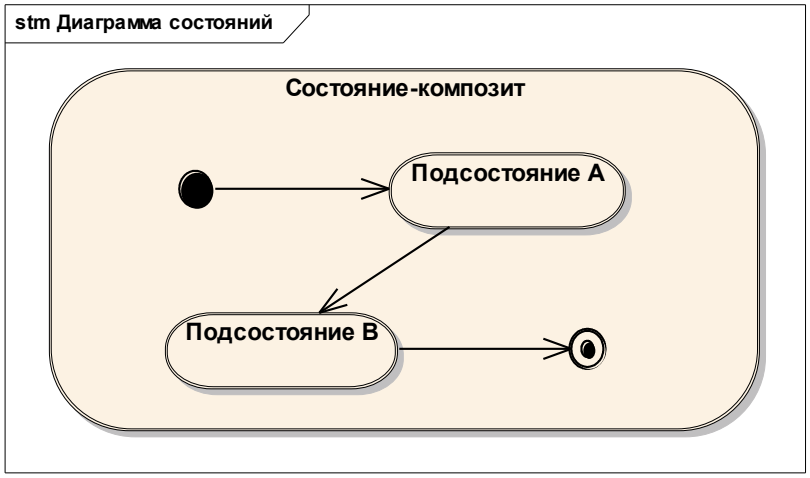


Рис. 3.10. Графическое представление составного состояния с последовательными подсостояниями

Составное состояние может содержать только последовательные или только параллельные подсостояния, каждое из которых в свою очередь также может являться состоянием-композитом и содержать внутри себя другие вложенные конечные автоматы. Количество уровней вложенности в языке UML не фиксировано.

Последовательные подсостояния (sequential substates) – вложенные состояния состояния-композита, в рамках которого в каждый момент времени объект может находиться только в одном подсостоянии.

В этом случае поведение объекта представляет собой последовательную смену подсостояний от начального вложенного состояния до конечного. Моделируемый объект или система продолжают находиться в составном состоянии, а использование последовательных вложенных состояний позволяет учесть более тонкие аспекты его внутреннего поведения.

Каждое составное состояние может содержать в качестве вложенных состояний начальное и конечное состояния. При этом начальное состояние является исходным, когда объект переходит в данное составное состояние, а переход в конечное подсостояние означает завершение нахождения объекта в данном составном состоянии. Важно отметить, что для последовательных подсостояний начальное и конечное состояния должны быть единственными в каждом составном состоянии.

В качестве примера на рис. 3.11 представлена часть модели состояний телефонного аппарата, а именно составное состояние «Дозвон до абонента», которое включает в себя два последовательные состояния: «Телефонная трубка поднята» и «Набор телефонного номера».

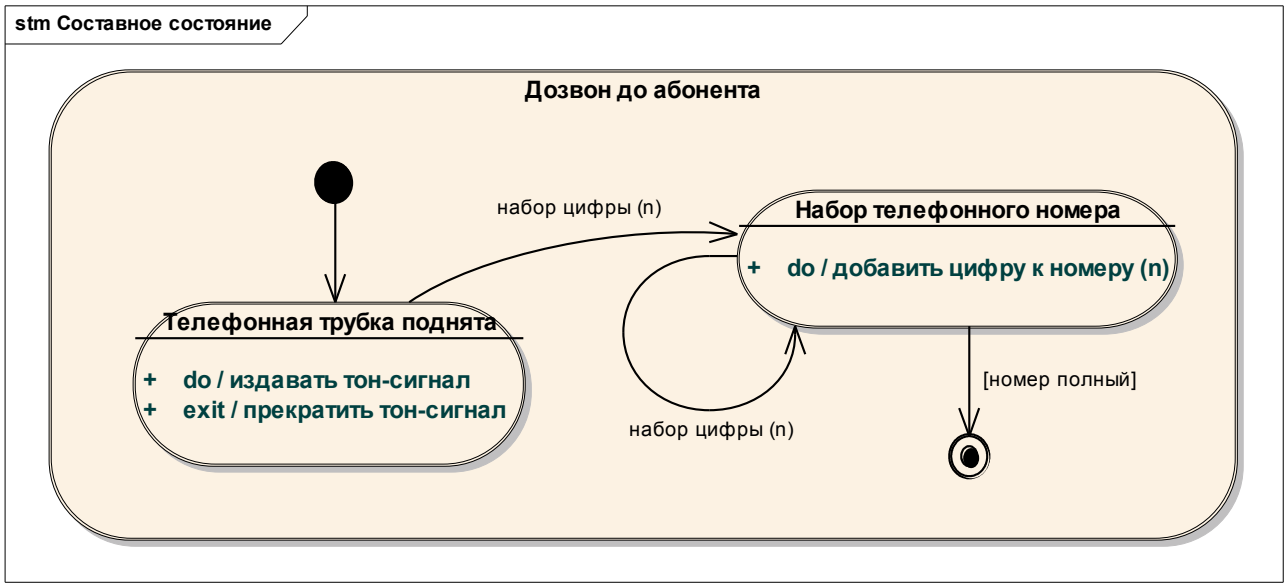


Рис. 3.11. Пример составного состояния с двумя вложенными последовательными подсостояниями

Параллельные подсостояния (concurrent substates) – вложенные состояния, используемые для спецификации двух и более конечных автоматов, которые могут выполняться параллельно внутри составного состояния.

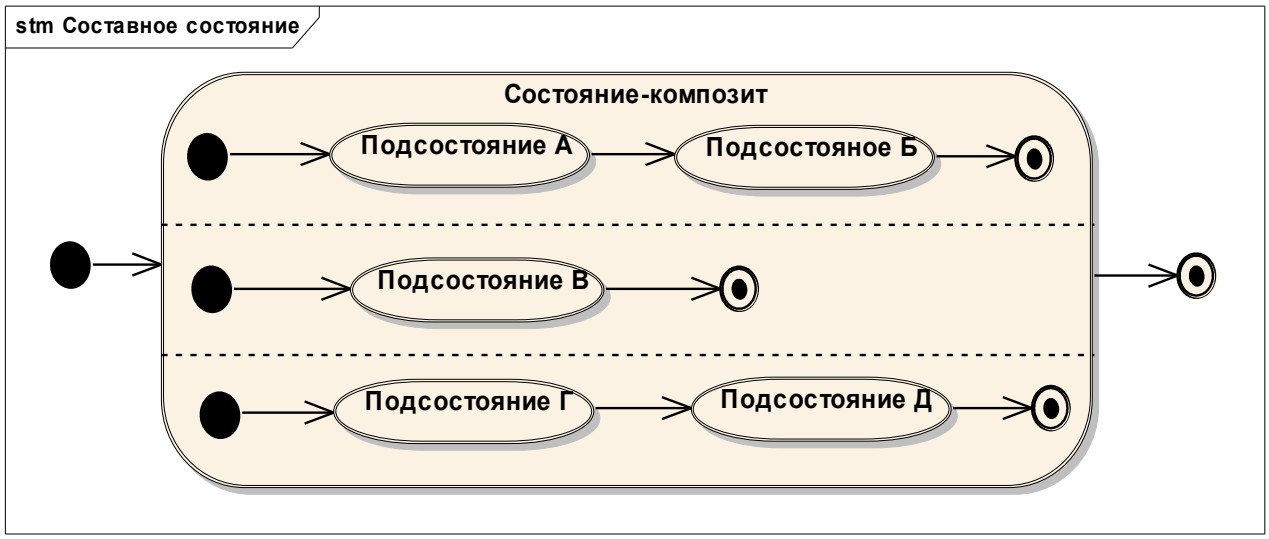


Рис. 3.12 Графическое изображение состояния композита с вложенными параллельными состояниями

Отдельные параллельные состояния могут, в свою очередь, состоять из нескольких последовательных подсостояний. В этом случае моделируемый

объект может находиться только в одном из последовательных подсостояний каждого из параллельных вложенных автоматов. Например (рис. 3.12): (А, В, Г), (Б, В, Г), (А, В, Д) и т.п.

Так как каждый регион вложенного состояния специфицирует некоторый конечный подавтомат, то для каждого из вложенных подавтоматов могут быть определены начальное и конечное состояния. При переходе в данное составное состояние каждый из конечных автоматов оказывается в своем начальном состоянии, а далее происходит параллельное выполнение каждого из этих конечных подавтоматов.

Важно отметить, что выход из такого составного состояния возможен только в том случае, когда все параллельные подавтоматы будут находиться в своих конечных состояниях. Если какой-либо из вложенных конечных автоматов придет в свое финальное состояние раньше других, то он должен ожидать, пока и другие подавтоматы не придут в свои финальные состояния.

На рис. 3.13 представлен пример составного состояния «Прохождение курса», включающего в себя параллельные конечные автоматы, один из которых состоит из последовательных вложенных состояний «Лабораторная 1» и «Лабораторная 2». После того, как все параллельные подсостояния будут пройдены, система сможет перейти в состояние «Пройден».

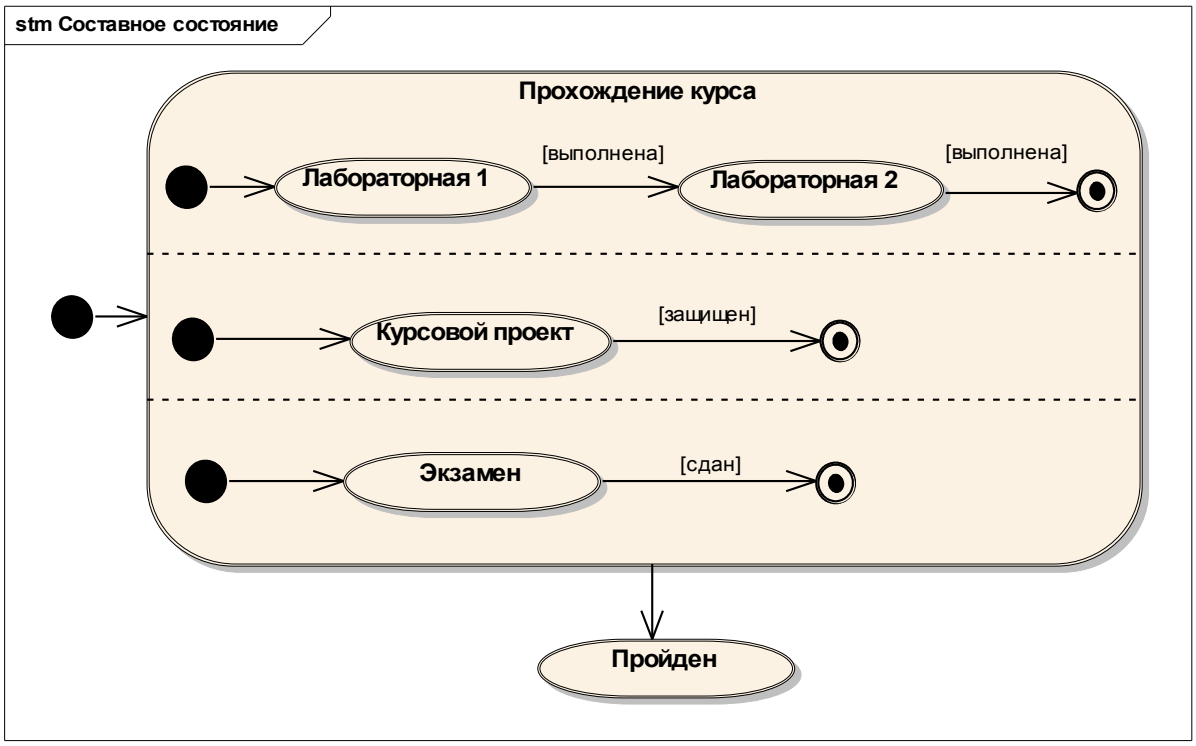


Рис. 3.13. Пример составного состояния с вложенными параллельными состояниями

При моделировании составных состояний можно использовать различные виды переходов в составное состояние и из составного состояния (рис. 3.14).

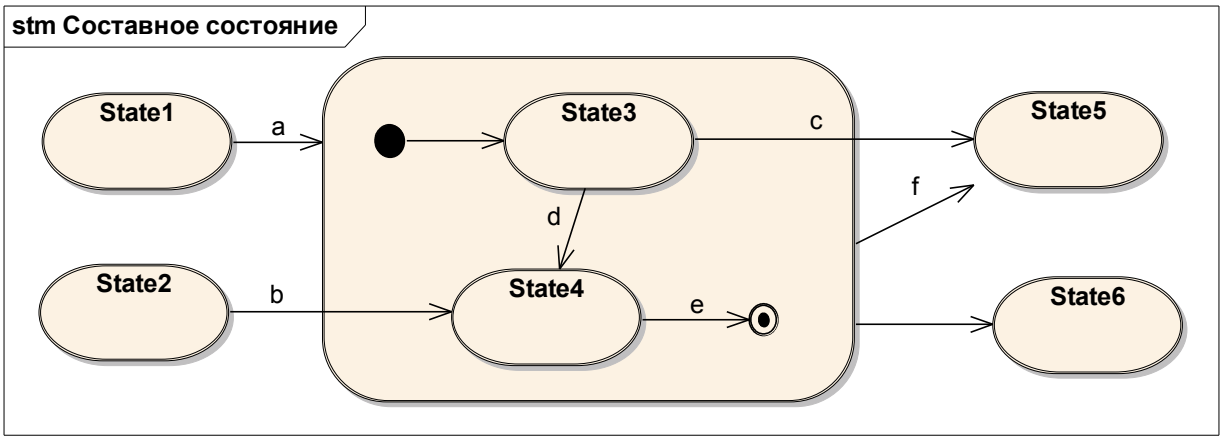


Рис. 3.14. Различные варианты переходов в составное состояние и из составного состояния

Переход, стрелка которого соединена с границей составного состояния, обозначает переход в это составное состояние. Он эквивалентен переходу в начальное состояние каждого из конечных подавтоматов, входящих в состав данного состояния-композита.

Переход, выходящий из составного состояния, так же относится к каждому из вложенных состояний. Такой переход означает, что моделируемая система, при наступлении события, вызвавшего данный переход, покидает составное состояние, независимо от того в каком из его подсостояний она находится в этот момент.

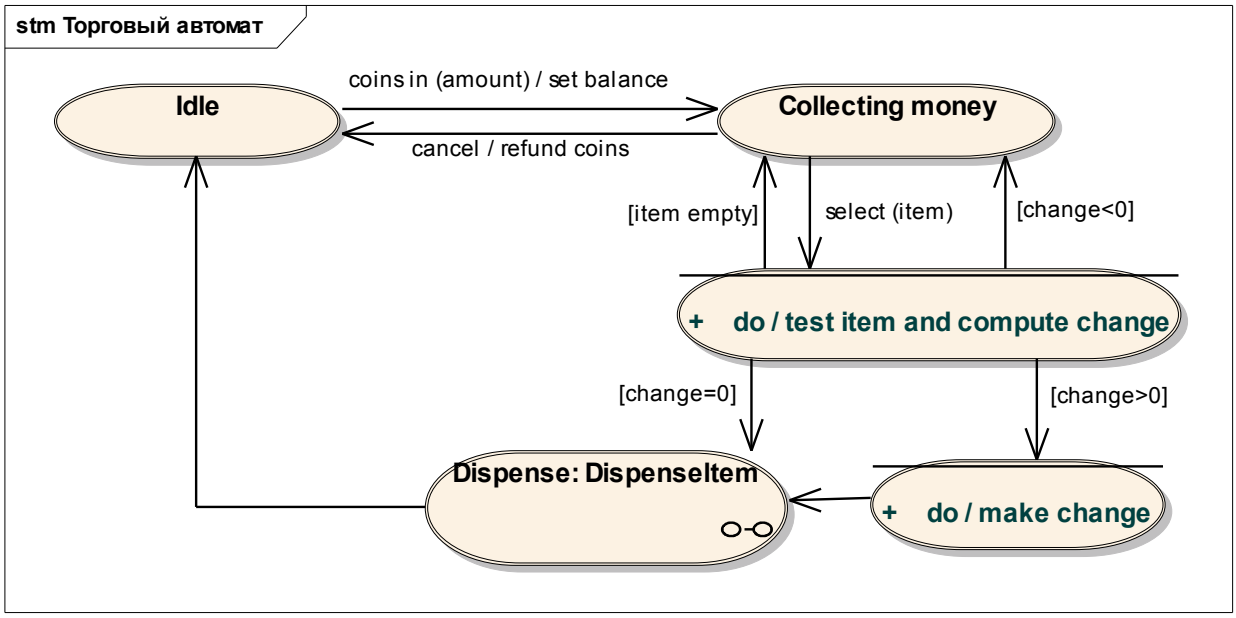
На диаграммах состояний также допускается изображение переходов, входящих в отдельное вложенное состояние, находящееся внутри состояния-композита. В этом случае стрелка перехода пересекает границу составного состояния и указывает на нужное вложенное состояние. Если требуется реализовать ситуацию, когда выход из отдельного подсостояния соответствует выходу из составного состояния, изображается переход, который непосредственно выходит из вложенного состояния и пересекает границу состояния – композита.

В простых задачах вложенные состояния можно реализовать на одной диаграмме с составными состояниями, как показано на предыдущих примерах. Но, если вложенный конечный автомат большой по масштабу и затрудняет визуализацию всей диаграммы состояний, тогда возникает необходимость скрыть внутреннюю структуру составного состояния.

В подобной ситуации внутренняя структура составного состояния не раскрывается на общей диаграмме, а изображается отдельно от основной диаграммы и может вызываться как часть составного состояния. В этом случае на исходной диаграмме составного состояния ставится специальный символ - пиктограмма, а также может указываться название состояния, после которого ставиться двоеточие и пишется название вложенного автомата.

На рис. 3.15а) приведен пример диаграммы состояний торгового автомата. Составное состояние «Dispense» (выдача) отмечено специальным символом в нижнем правом углу и в названии присутствует ссылка на вложенную диаграмму «DispenseItem».

а)



б)

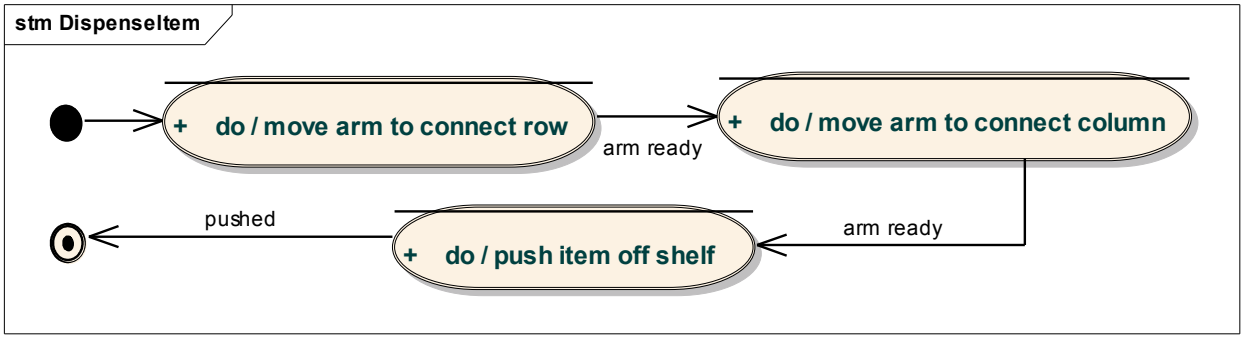


Рис. 3.15. Диаграмма состояний торгового автомата

Вложенная диаграмма «DispenseItem», которая представлена на рис. 3.15б), раскрывает состояние «Dispense» на дополнительной диаграмме и по необходимости может быть вызвана из основной диаграммы.

Итак, диаграммы состояний описывают комбинацию событий, состояний и переходов между ними для определенного класса, иначе говоря, этот тип диаграмм полностью описывает поведение объектов одного класса. Модель состояний состоит из множества диаграмм состояний – по одной на каждый класс из модели классов, поведение которого важно для проектируемой модели. Диаграммы состояний разных классов взаимодействуют друг с другом посредством общих событий, а их совокупность показывает возможное поведение системы в целом.

Упражнения

1. Разработайте диаграмму состояний для простейших электронных часов. Простейшие электронные часы состоят из дисплея и двух кнопок A и B. Часы могут работать в двух режимах: отображения и настройки. В режиме отображения часы показывают часы и минуты, между которыми мигает символ двоеточия.
   Режим настройки состоит из двух подрежимов: настройка часов и настройка минут. Кнопка A позволяет выбрать режим. Каждый раз при ее нажатии происходит переход к следующему режиму: отображение, установка часов, установка минут, отображение и т.д. Кнопка B позволяет увеличить значение часов или минут на единицу при каждом нажатии в одном из режимов установки. Чтобы кнопка могла породить новое событие, ее необходимо отпустить.
2. Измените диаграмму состояний из предыдущего примера, добавив ускоренную установку времени длительным нажатием кнопки B. Если кнопка B нажата и удерживается в течение пяти секунд в режиме установи времени, часы или минуты увеличиваются на единицу каждые полсекунды.
3. Спроектируйте диаграмму состояний для телефонного автоответчика. Автоответчик определяет входящий звонок по первому сигналу и отвечает заранее записанным сообщением. После этого автоответчик записывает сообщение звонящего. Когда звонящий вешает трубку, автоответчик тоже вешает трубку и отключается до следующего звонка
4. Измените диаграмму состояний из предыдущей задачи таким образом, чтобы автоответчик срабатывал по пятому сигналу вызова. Если кто-то подойдет к телефону до пятого звонка, автоответчик не должен ничего делать.
5. Разработайте диаграмму состояний протокола передачи данных. В персональном компьютере контроллер диска обычно передает поток байтов с дисковода в буфер памяти с помощью ведущего узла (центрального процессора или контроллера прямого доступа к памяти DMA). Контроллер передает ведущему узлу сигнал о каждом новом доступном байте. Данные должны быть считаны и сохранены для того, чтобы контроллер мог перейти к следующему байту. Когда контроллер обнаруживает, что данные были считаны, он сообщает об отсутствии данных до тех пор, пока не подготовит следующий байт. Если байт не будет считан до того, как контроллер подготовит следующий, контроллер выдает сигнал потери данных до тех пор, пока не получит сигнал сброса.
6. Смоделируйте диаграмму состояний для выделения и перетаскивания объектов при помощи редактора диаграмм (упражнение 8 из раздела «Моделирование классов»). Курсор управляется двухкнопочной мышью. При нажатии левой кнопки в момент нахождения курсора над объектом, объект выделяется. При этом выделение снимается с любого ранее

выделенного объекта. Если левая кнопка нажимается в тот момент, когда курсор не находится над объектом, то выделение снимается со всех ранее выделенных объектов. Перемещение мыши с нажатой левой кнопкой позволяет перетащить выделенный объект.

7. Разработайте диаграмму состояний для светофора на перекрестке. Одна пара фотоэлементов светофора контролирует полосы в направлении север-юг, из которых возможен поворот налево. Другая пара контролирует полосы в направлении запад-восток, из которых тоже возможен поворот налево. Сигнал светофора имеет два состояния: движение прямо и поворот налево, которые горят на светофоре определенное время. Если на одной из пар полос отсутствуют машины, управляющая логика светофора пропускает часть цикла, разрешающую левый поворот.
8. Разработайте диаграмму состояний копировального аппарата. В начальном состоянии копировальный аппарат выключен. Включение питания переводит аппарат в основное состояние: одна копия, автоматическая настройка контраста, нормальный размер. В процессе прогрева аппарат мигает индикатором готовности. Когда самопроверка аппарата завершается, индикатор готовности престает мигать и начинает гореть непрерывно. После этого аппарат считается готовым к работе. Оператор может изменить любой параметр, пока аппарат находится в режиме готовности. После установки нужных значений параметров оператор может начать копирование, которое продолжается пока не будет сделано необходимое количество копий. Исключительные ситуации связаны с застреванием бумаги или ее отсутствием. Если возникла исключительная ситуация, мигает индикатор ошибки до тех пор, пока либо не будет устранен затор бумаги, либо не будет добавлена новая пачка бумаги. После этого аппарат готов продолжить копирование.
9. Разработайте диаграмму состояний для задачи об интеренет-магазине (упражнение 9 из раздела «Моделирование классов»).

4. Моделирование взаимодействий

Модель взаимодействий – это третья составляющая моделирования систем, которая описывает взаимодействие объектов между собой. Моделирование взаимодействий включает в себя построение трех видов диаграмм: диаграмм вариантов использования, диаграмм последовательности и диаграмм деятельности. Другими словами, взаимодействия можно моделировать на разных уровнях абстрагирования.

На самом высоком уровне взаимодействие системы с внешними действующими лицами описывается вариантами использования. Каждый вариант использования описывает элемент функциональности, предоставляемой системой ее пользователям. Варианты использования полезны для представления в модели неформальных требований.

Диаграммы последовательности более детализированы, они показывают сообщения, которыми обмениваются объекты с течением времени. Диаграммы последовательности полезны для демонстрации последовательностей поведения, видимых пользователям системы. Наконец, диаграммы деятельности содержат всю информацию и показывают поток управления между этапами вычислений.

Итак, основным элементом разработки и планирования проекта является построение модели вариантов использования. Рассматриваемая модель описывает функциональное назначение системы в самом общем виде с точки зрения всех ее пользователей и заинтересованных лиц. Иначе говоря, модель вариантов использования должна отвечать на вопрос о том, что должна делать система в процессе своего функционирования, не затрагивая вопрос о том, как она должна это делать.

Графически модель вариантов использования или модель прецедентов проектируется при помощи диаграмм вариантов использования или диаграмм прецедентов.

4.1. Диаграммы вариантов использования

Основной задачей диаграмм вариантов использования является получение требований к системе от заказчика и пользователей. Привлекательность и эффективность этого вида диаграмм заключается в том, что они просты для понимания непрограммистами и в то же время достаточно формальны.

Диаграмма вариантов использования или прецедентов (use case diagram) – это граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров и отношений между этими элементами. При этом отдельные элементы диаграммы заключают в прямоугольник, который обозначает границы проектируемой системы.

Отношения, которые могут быть изображены на данном графе, представляют собой только фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

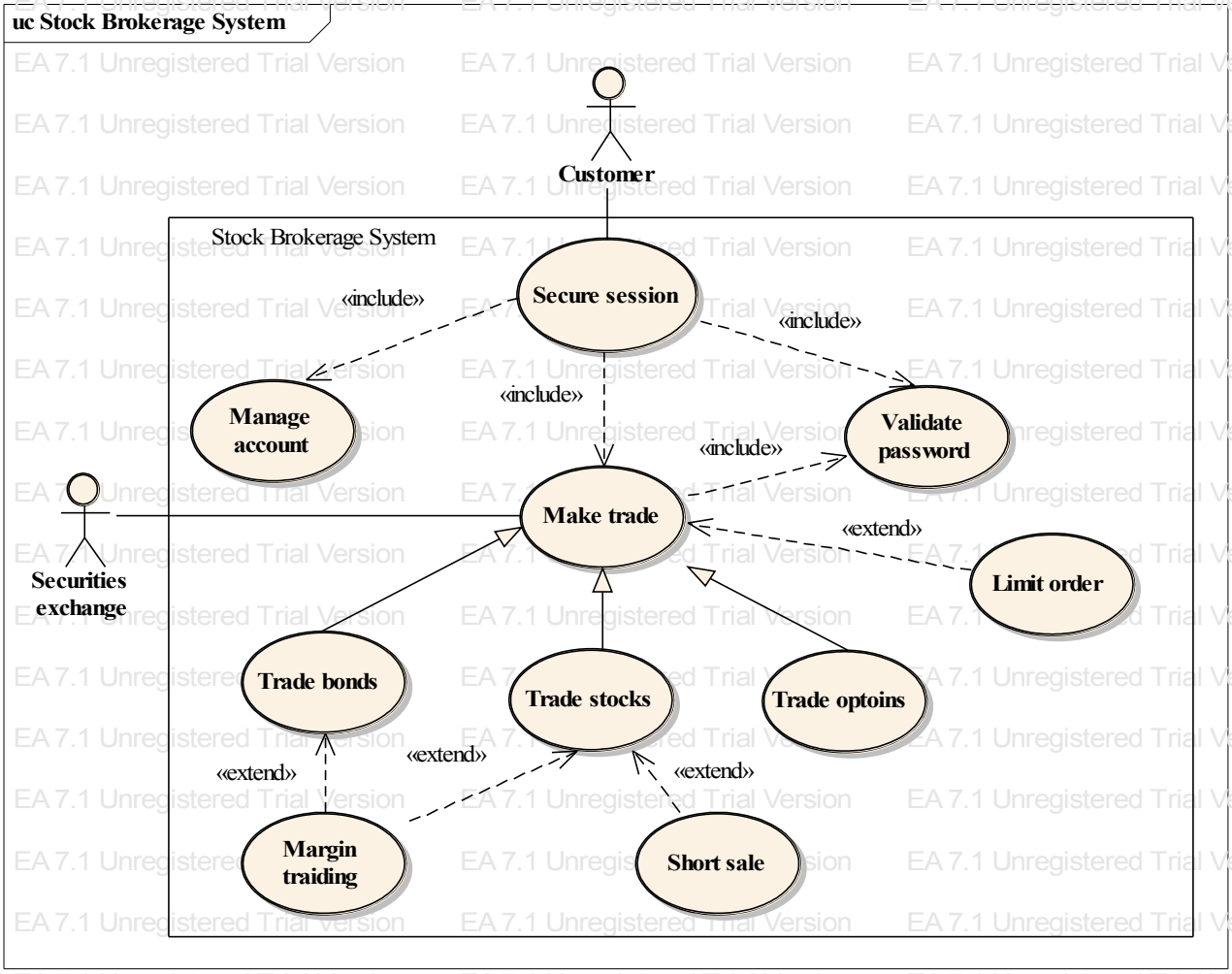


Рис. 4.1. Пример диаграммы вариантов использования (сетевая брокерская система)

Базовыми элементами диаграммы вариантов использования являются актеры и собственно сами варианты использования.

4.1.1. Актеры и варианты использования

Любые, в том числе и программные системы, проектируются с учетом того, что в процессе своей работы они будут использоваться людьми или взаимодействовать с другими системами.

Сущности, с которыми взаимодействует система в процессе своей работы, называются актерами.

Актер (actor) - это любая внешняя по отношению к проектируемой системе сущность, которая взаимодействует с данной системой.

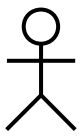
Другими словами, актер (действующее лицо) - это непосредственный внешний пользователь системы. Это объект или множество объектов, взаимодействующих с системой, но не являющихся ее частью.

Важно отметить, что каждое действующее лицо или актер должно иметь одну четко определенную цель, то есть каждое действующее лицо является обобщением группы объектов, ведущих себя определенным образом по отношению к системе. Если некоторая внешняя сущность выступает по отношению к системе в нескольких ролях, то она должна быть изображена на диаграмме в виде нескольких актеров. Например, владелец персонального компьютера может устанавливать программное обеспечение, настраивать базу данных и отправлять электронную почту. Эти функции существенно отличаются друг от друга и должны быть разделены между тремя действующими лицами: системным администратором, администратором базы данных и обычным пользователем.

Поскольку актер всегда находится вне проектируемой системы, его внутренняя структура в модели никак не определяется. Для актера важно только его внешнее представление, то есть, как он воспринимается со стороны моделируемой системы.

Графически действующие лица или актеры изображаются в виде фигурки человечка, которую так же называют «проволочный человечек», под которой пишется имя актера.

Имя актера должно начинаться с большой буквы и быть достаточно информативным с точки зрения семантики. Не следует давать актерам имена собственные, потому что, как уже было сказано ранее, одна и та же сущность может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы.



Клиент

Рис. 4.2. Графическое изображение актера

Несмотря на «человеческий вид» этого обозначения, не следует забывать, что актеры – это не обязательно люди. Для моделируемой программной системы актерами могут быть как непосредственные пользователи, так и другие программные, организационные, информационные или аппаратные системы. Все они являются внешними сущностями по отношению к системе и представляются в виде актеров.

Различные взаимодействия действующих лиц с системой группируются в варианты использования.

Вариант использования (use case) или прецедент – это внешняя спецификация последовательности действий, которые система может выполнять в процессе взаимодействия с актерами.

Вариант использования представляет собой описание общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Несмотря на то, что каждый вариант использования предполагает реализацию системой некоторой последовательности действий, которые должны быть выполнены с целью предоставления соответствующего результата или сервиса, сами эти действия на диаграмме вариантов использования никак не отображаются.

Цель спецификации отдельного варианта использования заключается в том, чтобы зафиксировать фрагмент поведения или функциональности проектируемой системы без указания особенностей его реализации.

Графическим изображением варианта использования является эллипс. Внутри эллипса пишется имя варианта использования, которое формулируется в виде законченного предложения.

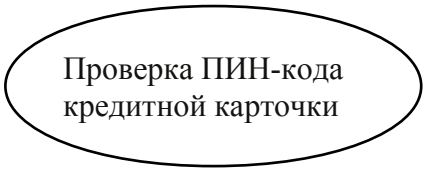


Рис. 4.3. Графическое изображение варианта использования

Совокупность всех вариантов использования, рассматриваемых в контексте проектируемой системы, заключается в границу описываемой системы или образует ее субъект.

Для изображения субъекта на диаграмме, варианты использования системы заключаются в прямоугольник, снаружи которого изображаются действующие лица. Название системы (субъекта) может быть указано с большой буквы около одного из краев прямоугольника.

На рис. 4.4 приведен пример диаграммы вариантов использования торгового автомата. Субъект «Торговый автомат» образован совокупностью вариантов использования, которые описывают поведение системы при взаимодействии с различными внешними пользователями.

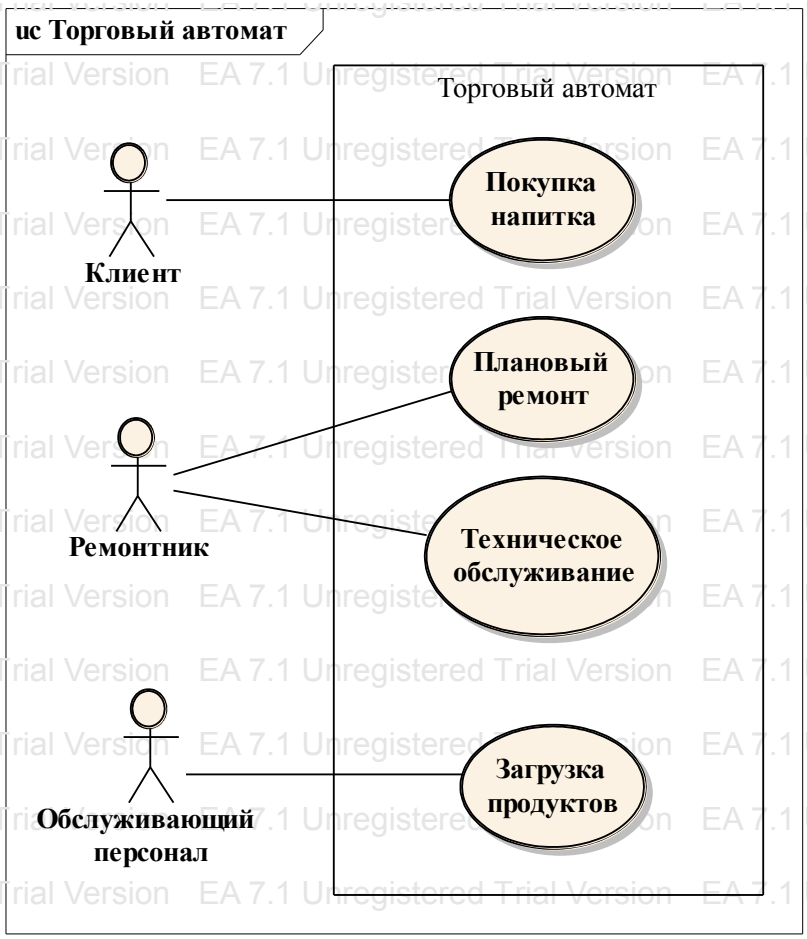


Рис. 4.4. Варианты использования торгового автомата

Таким образом, на диаграммах вариантов использования (прецедентов) каждый вариант использования (прецедент) удобно рассматривать как отдельный сервис, который субъект предоставляет по запросу тех или иных актеров. Каждый такой сервис определяет один из возможных способов применения системы по своему целевому назначению. Другими словами отдельный вариант использования представляет собой последовательность действий, выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом, то есть описывает типичное взаимодействие между пользователем и системой.

Важно отметить, что отдельный вариант использования должен объединять все поведение, имеющее отношение к элементу функциональности системы, то есть он не должен обладать слишком узким определением. Например, «Набрать телефонный номер» – не самый подходящий вариант использования для телефонной системы. Это всего лишь часть варианта использования «Позвонить», который включает в себя вызов, разговор и завершение вызова.

4.2. Отношения на диаграммах вариантов использования

Между элементами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимосвязи экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. Это значит, что актер обращается к нескольким сервисам данной системы. В свою очередь, один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них один сервис. В то же время два варианта использования, определенные в рамках одной системы, могут взаимодействовать друг с другом, однако характер этих взаимодействий будет отличаться от взаимодействия с актерами.

В языке UML определено несколько стандартных видов отношений, используемых на диаграммах вариантов использования:

- ассоциации *(association);*
- включения *(include);*
- расширения *(extend);*
- обобщения *(generalization).*

4.2.1. Отношения между актерами и вариантами использования

На диаграммах вариантов использования актеров и варианты использования связывает уже знакомое нам по диаграммам классов отношение ассоциации.

Отношение ассоциации (association) в контексте диаграммы вариантов использования применяется для обозначения взаимодействия актера с вариантом использования. Такая ассоциация может указывать на то, что актер инициирует соответствующий вариант использования или данному актеру предоставляется информация о результатах функционирования моделируемой системы.

Графически ассоциация изображается сплошной линией с необязательным символом навигации (стрелкой) на конце. Если направление взаимодействия не является для разработчика принципиальным, то может быть использована ненаправленная ассоциация, обозначаемая сплошной линией без стрелок. Если направление имеет значение, то используется направленная ассоциация, которая обозначается сплошной линией со стрелкой, указывающей направление ассоциации.

Один вариант использования может иметь несколько ассоциаций с несколькими актерами и наоборот, один актер может иметь ассоциации с несколькими вариантами использования.

В общем случае отношение ассоциации может иметь собственное имя, кратность, имена полюсов ассоциации. Однако эти характеристики на диаграммах вариантов использования практически не специфицируются.

Пример графического представления ассоциации рассмотрен на рис. 4.4. Например, актер «Клиент» связан отношением ассоциации вариантом использования «Покупка напитка». Это означает, что данный сервис предоставляется только одному актеру, который является «Клиентом». Актер «Ремонтник» связан ассоциациями с двумя вариантами использования, то есть он может инициировать два варианта поведения данной системы.

4.2.2. **Отношения между вариантами использования**

Независимых вариантов использования может быть достаточно для описания простых приложений. Однако крупные приложения требуют структурирования вариантов использования. Составные варианты использования можно конструировать из более простых вариантов при помощи отношении включения, расширения и обобщения.

Отношение включения (include) – описывает ситуацию, когда некоторый вариант использования содержит поведение, определяемое в другом варианте использования. Иначе говоря, это отношение, которое позволяет включить последовательность поведения одного варианта использования в другой.

Отношение включения является бинарным, то есть может связывать только два и не более элементов модели. Данное отношение применяется в тех ситуациях, когда имеется какой либо фрагмент поведения системы, который повторяется более чем в одном варианте использования.

Графически отношение включения обозначается пунктирной линией со стрелкой, направленной от исходного (включающего) варианта использования к целевому (включаемому) варианту использования. Над стрелкой ставиться ключевое слово *«include».*

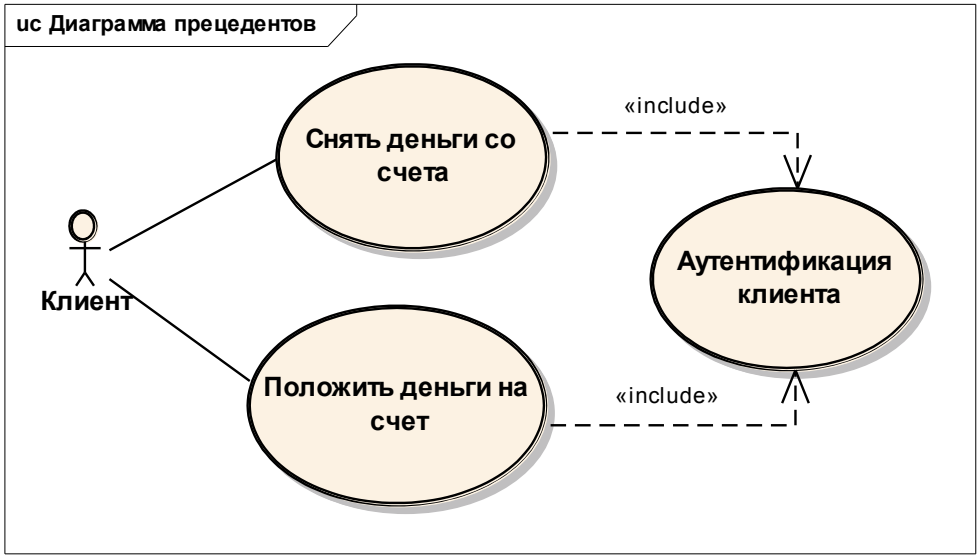


Рис. 4.5. Графическое изображение отношения включения

На рис. 4.5 приведен пример, в котором рассмотрено применение отношения включения между вариантами использования. Варианты использования «Снять деньги со счета» и «Положить деньги на счет» всегда включают в свое поведение выполнение варианта использования «Аутентификация клиента». Это значит, что при попытке снять или положить деньги на счет, всегда будет проводиться запрос и проверка пароля клиента на доступ в данную систему. Другими словами поведение варианта использования «Аутентификация клиента» является частью поведения двух других вариантов использования, представленных на данной диаграмме.

Следует помнить, что на одной диаграмме вариантов использования не может быть замкнутого цикла по отношению включения между вариантами использования. Иначе диаграмма вариантов использования будет не согласованной, тем более что автоматизированные средства разработки программного обеспечения (CASE- средства) не позволяют выявить такого рода ошибки.

Отношение расширения (extend) – определяет взаимосвязь одного варианта использования с другим, поведение которого задействуется первым вариантом не всегда, а только при выполнении некоторых дополнительных условий. Отношение расширения моделирует необязательное поведение системы – это просто отдельные последовательности действий, выполняемые при определенных обстоятельствах. Проще говоря, данное отношение добавляет к варианту использования дополнительное поведение.

Отношение расширения аналогично отношению включения, рассматриваемому с противоположной точки зрения. В случае расширения один вариант использования (расширяющий) добавляет себя к другому варианту использования (базовому), тогда как при включении один вариант использования явным образом включает другой. Расширение описывает типичную ситуацию, когда изначально определяются некоторые возможности системы, к которым затем добавляются новые функции.

И отношение расширения, и отношение включения добавляют поведение к базовому варианту использования. Однако при использовании отношения расширения, базовый вариант использования может быть вполне самостоятельным, то есть он может выполняться и без расширений.

Графически отношение расширения обозначается пунктирной линией со стрелкой, направленной от расширяющего варианта использования к базовому. Над стрелкой ставится ключевое слово *«extend».*

В приведенном на рис. 4.6 примере рассмотрено применение отношения расширения между вариантами использования «Оформление покупки» и «Предоставление скидки постоянному клиенту», то есть если покупатель является постоянным, то ему может быть предоставлена скидка. При этом сам вариант использования «Оформление покупки» выполняется независимо от варианта использования «Предоставление скидки».

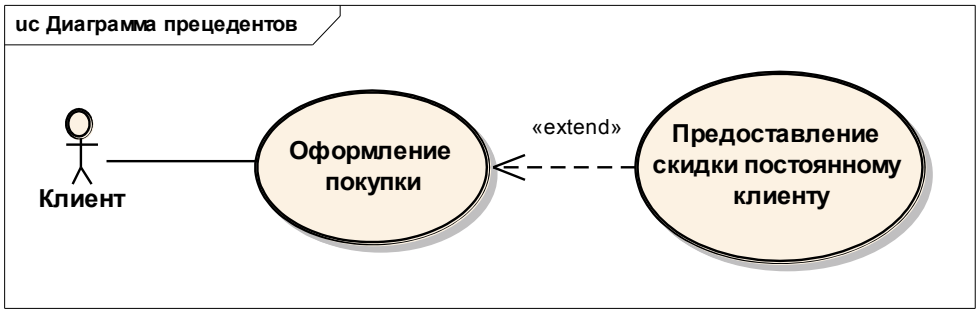


Рис. 4.6. Графическое изображение отношения расширения

Однако в приведенном примере не видно, при каких именно условиях клиенту может быть предоставлена скидка, то есть для того, чтобы расширение имело место должно быть выполнено специальное условие для данного расширения, например наличие дисконтной карты у покупателя.

В таких случаях отношение расширения может снабжаться некоторым прикрепленным к нему условием, которое должно определять будет выполнено расширение данного варианта использования или нет.

Для моделирования подобных ситуаций используется ***точка расширения (extension point)***, в которой происходит проверка условий расширения и подключаются действия из расширяющих вариантов использования.

Точку расширения можно специфицировать следующим образом. К соответствующему отношению расширения присоединяется ***примечание (note)***, которое изображается в виде прямоугольника с «загнутым» правым верхним углом («собачье ухо»).

Примечание должно содержать запись условия в форме ограничения, заключенного в фигурные скобки. Ниже строки с условием после ключевого слова «*extension point*» и двоеточия записывается имя точки расширения, которое также должно быть указано и в базовом варианте использования после ключевого слова. Пример использования точки расширения приведен на рис.4.7.

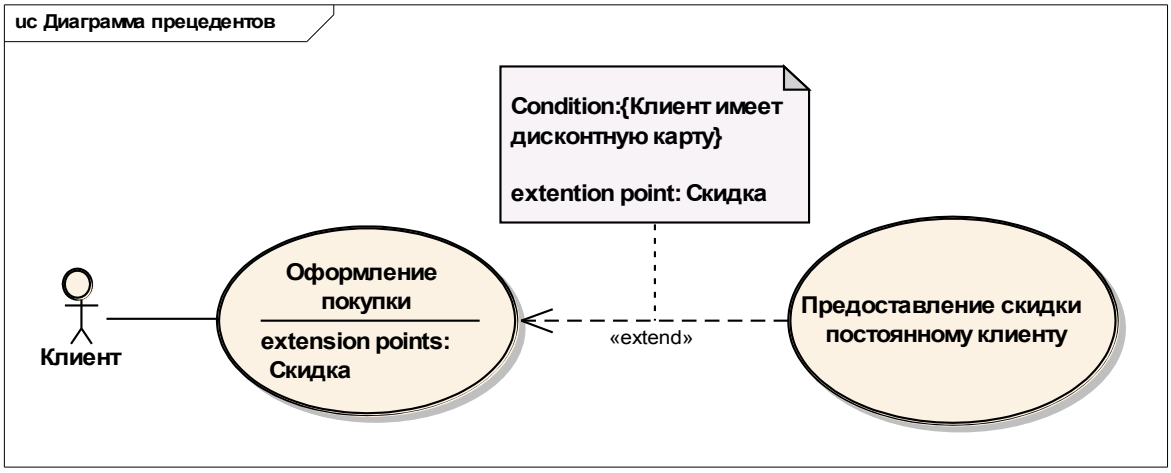


Рис. 4.7. Графическое изображение отношения расширения с использованием точки расширения.

Вариант использования может иметь сколь угодно много точек расширения. Комментарий позволяет сопоставить конкретный расширяющий вариант использования с определенной точкой расширения, так как в нем указывается и условие расширения и точка расширения, в которой происходит подключение расширяющего варианта использования. Если точка расширения единственная и из контекста понятно, какое условие расширения должно выполняться, то описание можно опустить.

В общем случае следует помнить, что поведение любого базового класса не должно зависеть от поведения его расширений и на одной диаграмме вариантов использования не может быть замкнутого пути по отношению расширения.

Отношение обобщения (generalization) - предназначено для спецификации случая, когда один элемент модели является специальным или частным случаем другого элемента модели. Данное отношение позволяет описывать вариации базового варианта использования, как и обобщение для классов. Вариант использования, являющийся предком, описывает общую последовательность поведения. Его потомки включают в себя поведение варианта использования, являющегося предком и конкретизируют поведение предка, добавляя новые элементы в его последовательность поведения или уточняя существующие.

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов. При этом потомки участвуют во всех отношениях родительских вариантов использования. Иначе говоря, если вариант использования встречается в модели в нескольких разновидностях, следует выделить общее поведение в базовый класс, а затем конкретизировать каждую из разновидностей. Следует отметить, что не стоит применять обобщение только для того, чтобы описать общий фрагмент поведения, так как для этого предназначено отношение включения.

Почти во всех отношениях обобщение вариантов использования ведет себя так же, как и обобщение классов. Но существует и различие. Подкласс может добавлять собственные атрибуты к атрибутам суперкласса, но порядок этих атрибутов значения не имеет. Вариант-потомок также добавляет элементы поведения, однако они должны присутствовать в правильных местах внутри последовательности элементов поведения его предка.

Графически отношение обобщения изображается в виде линии с не закрашенной стрелкой, которая указывает на вариант-предок. Для этого отношения применяется та же система обозначений, что и на диаграммах классов.

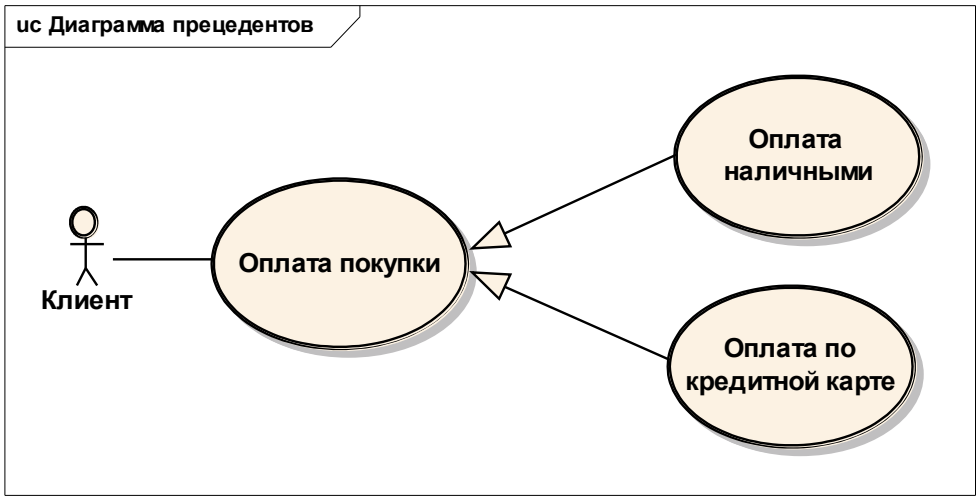


Рис. 4.8. Графическое изображение отношения обобщения

В приведенном примере (рис. 4.8), вариант использования «Оплата покупки» является вариантом-предком по отношению к вариантам «Оплата по кредитной карте» и «Оплата наличными», так как они являются его разновидностью.

4.2.3. Отношения между актерами

Актеры могут быть связаны между единственно допустимым отношением – ***отношением обобщения (generalize).*** Это отношение используется, если два и более актера имеют общие свойства, то есть могут взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом.

Данное отношение является направленным и указывает на факт специализации одних актеров относительно других.

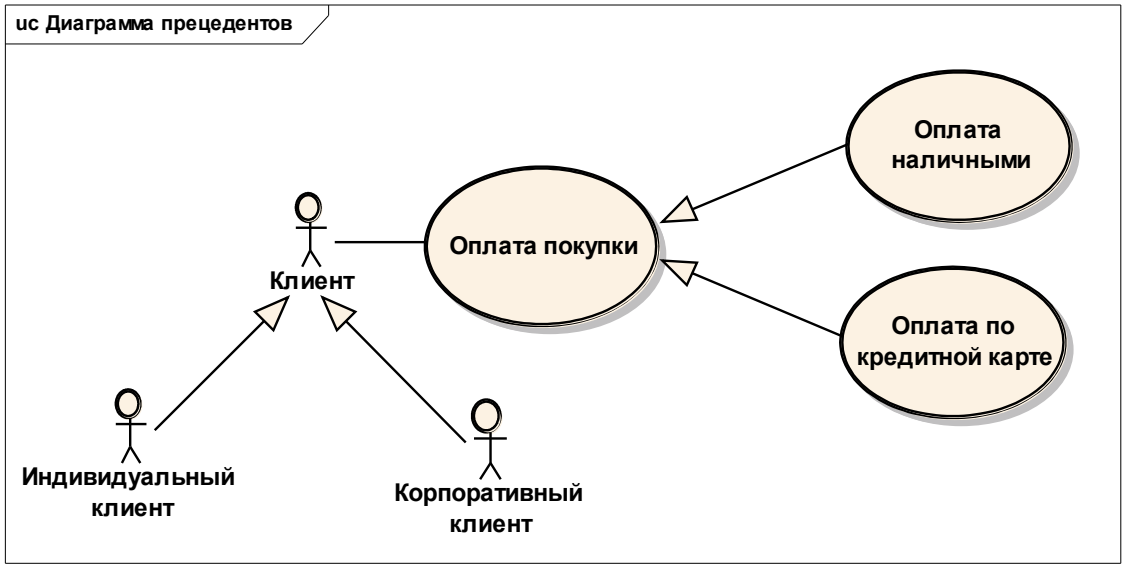


Рис. 4.9. Пример применения отношения обобщения между актерами.

В приведенном на рис. 4.9 фрагменте диаграммы отношение обобщения служит для указания того факта, что актеры корпоративный и индивидуальный являются частными случаями актера клиент.

4.3. Дополнительные спецификации вариантов использования.

Следует отметить, что одним из требований языка UML является самодостаточность диаграмм для представления информации о моделях проектируемых систем. Однако большинство разработчиков и экспертов отмечают, что изобразительных средств языка UML явно не хватает для того, чтобы учесть на диаграммах вариантов использования особенности поведения сложной системы. С этой целью рекомендуется дополнять этот тип диаграмм текстовыми сценариями, которые уточняют или детализируют последовательность действий, совершаемых системой при выполнении ее вариантов использования.

Сценарий (scenario) – определенная последовательность действий, которая описывает действия актеров и поведение моделируемой системы в форме обычного текста, то есть сценарий – это описание потока действий или событий в текстовой форме, позволяющее дополнительно специфицировать поведение сложной системы.

Некоторые варианты использования характеризуются фиксированной последовательностью событий. Однако чаще последовательность событий может варьироваться в определенных пределах. Проектировщик должен учитывать все возможные последовательности действий (сообщений). Это значит, что устойчивая система должна обрабатывать все разновидности поведения, то есть вариант использования объединяет все поведение, имеющее отношение к функциональности системы. Обычно сначала определяется основная последовательность, а затем – необязательные последовательности, повторения, сбойные и исключительные ситуации, отмены запросов и другие вариации.

Таким образом, каждый вариант использования, представленный на диаграмме, должен иметь связанный с ним сценарий, который включает в себя следующие разделы:

- ***Описание.*** Каждый вариант использования должен иметь связанное с ним короткое описание того, что он будет делать. Оно должно определять типы пользователей, выполняющих вариант использования, и ожидаемый ими конечный результат.
- ***Действующие лица.*** Определяет действующие лица, участвующие в варианте использования.

- ***Предусловия.*** Это условия, которые должны быть выполнены, прежде чем вариант использования начнет выполняться сам. Предварительные условия бывают не у всех вариантов использования.
- ***Основной поток событий.*** Или нормальный поток событий - поэтапно описывает, что должно происходить во время выполнения заложенной в вариант использования функциональности, то есть что будет делать система с точки зрения пользователя
- ***Альтернативный поток событий.*** Или исключения - описывают отклонения от основного потока, а также потоки ошибок.
- ***Постусловия.*** Это условия, которые должны быть выполнены после завершения варианта использования.

Сценарии могут быть записаны в различных формах: структурированный текст, псевдокод, таблица. Выбор способа записи сценария произволен. Главное, чтобы сценарий в последовательной форме описывал завершенное конкретное взаимодействие, имеющее с точки зрения пользователя определенную цель.

Для иллюстрации особенностей спецификации вариантов использования рассмотрим модель торгового автомата, представленного на рис. 4.4.

Субъект «Торговый автомат» включает следующие варианты использования:

- **Купить напиток.** Торговый автомат выдает напиток после того, как клиент выбирает нужный вариант и платит за него.
- **Провести плановый ремонт.** Ремонтник выполняет плановое техобслуживание автомата необходимое для обеспечения его безотказной работы.
- **Провести техническое обслуживание.** Ремонтник выполняет незапланированное обслуживание автомата при выходе его из строя.
- **Загрузить продукты.** Обслуживающий персонал загружает продукты в торговый автомат для пополнения запасов продаваемых напитков.

Приведем примерный сценарий для описания варианта использования ***«Покупка напитка»***.

Вариант использования: Покупка напитка.

Краткое описание: Торговый автомат выдает напиток после того, как клиент выбирает нужный вариант и платит за него.

Действующие лица: Клиент.

Предусловия: Автомат ожидает опускания монеты.

Основной поток событий: Автомат изначально находится в состоянии ожидания и выводит на дисплей сообщение «Опустите монеты». Клиент опускает монеты в автомат. Автомат выводит на дисплей принятую от клиента сумму и включает подсветку кнопок с названиями напитков. Клиент выбирает напиток. Автомат выдает соответствующий напиток и сдачу, если напиток стоит меньше, чем заплатил клиент.

Альтернативный поток событий (исключения):

Отмена: Если клиент нажмет кнопку отмены до того, как произведет выбор напитка, автомат вернет деньги и перейдет в состояние ожидания

Напиток отсутствует: Если клиент выбирает напиток, который в данный момент отсутствует в автомате, выводится сообщение «Напиток отсутствует». Автомат готов к приему монет и выбору напитка клиентом.

Недостаточно денег: Если клиент выбирает напиток, который стоит больше, чем он заплатил, выводится сообщение « Необходимо доплатить ** . ** руб.», где ** . ** - недостающая сумма. Автомат готов к приему монет и выбору напитка.

Нет сдачи: Если клиент заплатил больше чем стоит выбранный напиток, а в автомате нет денег, чтобы правильно выдать сдачу. Выдается сообщение «Невозможно выдать сдачу». Автомат готов к приему денег и выбору напитка.

Постусловия: Автомат готов к приему монет.

Данный сценарий может быть помещен в текстовый документ, прикрепленный к диаграмме вариантов использования, либо записан в спецификации к варианту использования, которая применяется в автоматической программной системе, используемой для моделирования.

Сценарий к варианту использования можно оформить в виде таблицы, которую также можно рассматривать как шаблон для написания сценария.

<table>
<tr><th>Главный раздел</th><th colspan="2">Раздел «Основной ход событий»</th><th>Раздел «Исключения»</th><th>Раздел «Примечания</th></tr>
<tr><td rowspan="5">Имя варианта использования

Актеры

Цель

Краткое описание

Тип

Ссылки на другие варианты использования</td><td>Действия актеров</td><td>Отклик системы</td><td>Исключение № 1</td><td>Примечание № 1</td></tr>
<tr><td colspan="2" rowspan="4">Типичный ход событий, приводящий к успешному выполнению данного варианта использования</td><td>Исключение № 2</td><td>Примечание № 2</td></tr>
<tr><td>…</td><td>…</td></tr>
<tr><td>Исключение № n</td><td>Примечание № n</td></tr>
</table>

Для удобства чтения и записи сценария можно разбить таблицу на разделы и располагать их последовательно сверху вниз. При этом написание

сценария начинают с основных вариантов использования, после чего рассматриваются сценарии второстепенных или включаемых прецедентов.

Сценарии расширяющих вариантов использования помещаются в раздел исключений базового варианта использования. Желательно, чтобы каждому исключению соответствовал отдельный вариант использования, который соединяется с базовым отношением расширения.

Итак, основной задачей диаграмм вариантов использования (прецедентов) является получение требований к проектируемой системе от заказчика и пользователей. С одной стороны, данный тип диаграмм может служить основой для согласования с заказчиком функциональных требований к системе на ранних стадиях проектирования, а с другой диаграммы вариантов использования позволяют разработчикам понять назначение элементов системы и лучше представить себе их поведение.

4.4. Диаграммы последовательности

Диаграммы последовательности представляют собой углубленное рассмотрение ситуаций, описываемых вариантами использования. Каждый вариант использования требует, по крайней мере, одной диаграммы последовательности, которая отражает поток событий, происходящих в рамках данного прецедента. С помощью диаграмм последовательности можно представить взаимодействие элементов модели как своеобразный временной график «жизни» всей совокупности объектов, связанных между собой какими-либо отношениями для реализации конкретного варианта использования программной системы.

Диаграмма последовательности (sequence diagram) – это диаграмма, на которой показаны взаимодействия элементов системы, упорядоченные по времени их появления.

Основной задачей диаграмм последовательности является спецификация участников взаимодействия и последовательности сообщений, которыми они обмениваются с течением времени.

Каждая диаграмма последовательности показывает конкретную последовательность поведения прецедента, то есть диаграммы данного типа описывают взаимодействие между объектами системы в процессе полного или частичного выполнения варианта использования. Лучше всего изображать на диаграмме определенную часть поведения проектируемой системы и не стремиться к максимальной общности.

На диаграмме последовательности можно показывать условия переходов между потоками управления в процессе выполнения данного варианта использования, но обычно модель получается яснее, если для каждого существенного потока управления строится своя диаграмма последовательности. Иначе говоря, для каждой задачи, являющейся

составляющей сеанса работы моделируемой системы, можно спроектировать отдельную диаграмму последовательности.

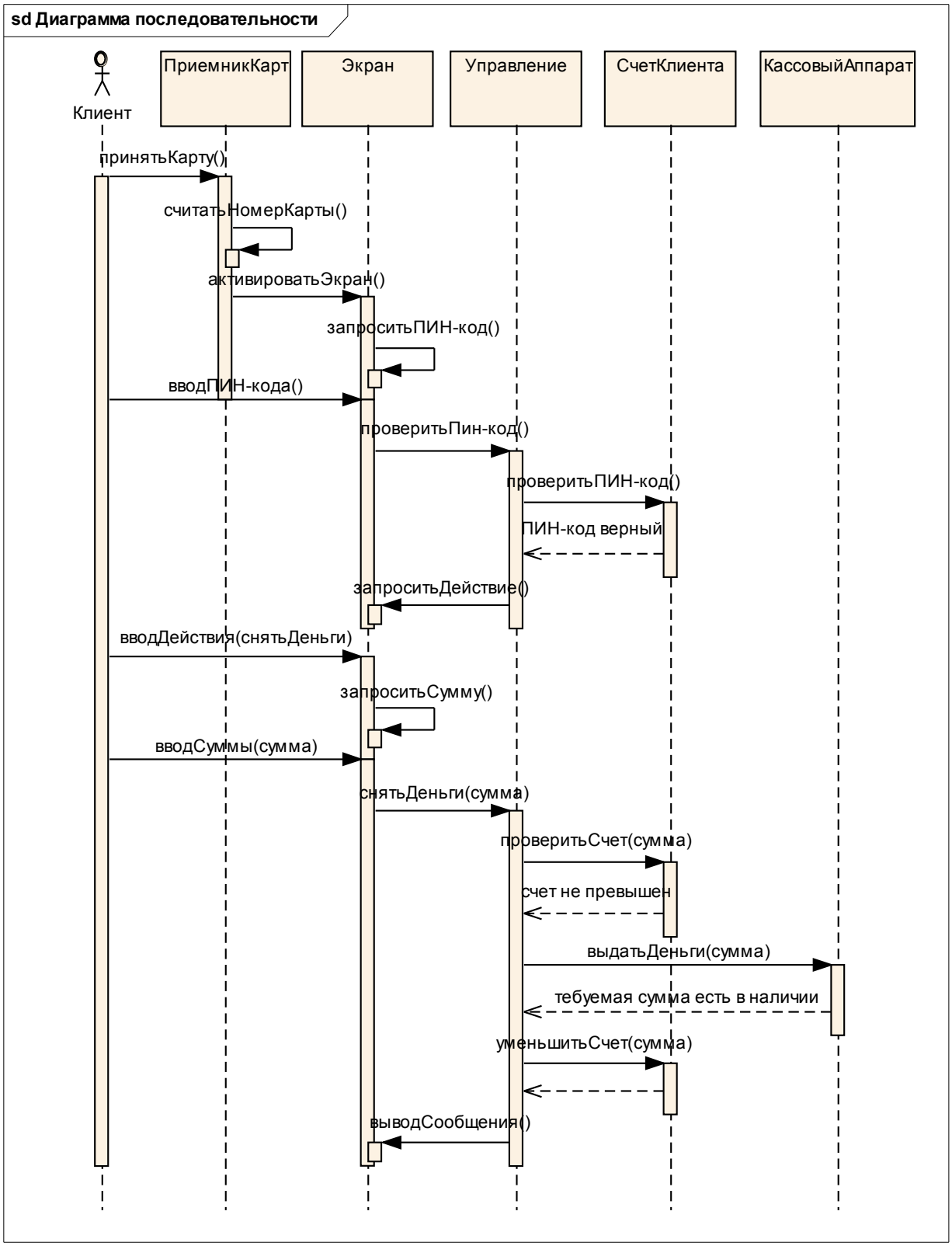


Рис. 4.10. Пример диаграммы последовательности (для основного потока событий варианта использования «Снятие денег через банкомат»).

Следует отметить, что диаграмма последовательности строится обязательно для каждой исключительной ситуации, возможной для данного варианта использования.

Графически диаграмма последовательности имеет два измерения. Одно – слева направо в виде вертикальных линий, каждая из которых соответствует линии жизни отдельного участника взаимодействия. Второе – вертикальная временная ось, направленная сверху вниз.

Начальному моменту времени соответствует самая верхняя часть диаграммы. Масштаб для оси времени на диаграмме последовательности не указывается, поскольку эта диаграмма предназначена для моделирования только лишь временного порядка следования сообщений типа «раньше – позже».

Диаграмма последовательности имеет два основных элемента – это *линия жизни объекта и сообщения,* которыми обмениваются объекты в процессе взаимодействия.

4.4.1. Линия жизни объекта

Линия жизни объекта (life line) – специфицирует одного отдельного участника взаимодействия или отдельную взаимодействующую сущность. Это вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода времени.

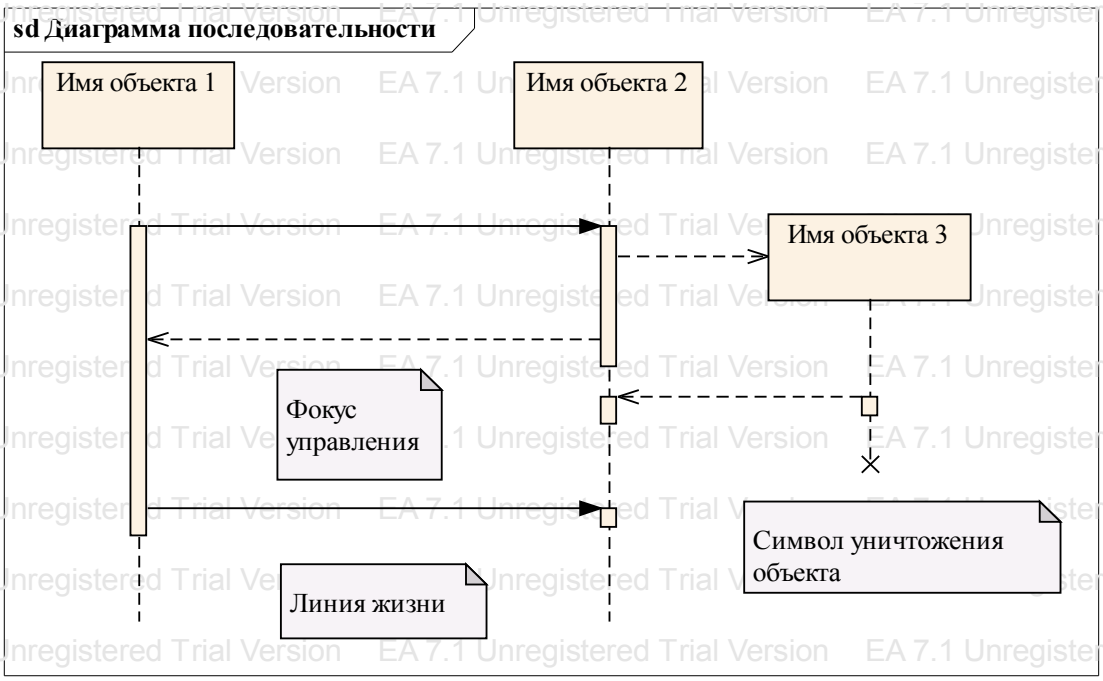


Рис. 4.11. Графические элементы диаграммы последовательности

Каждый объект на диаграмме последовательности изображается в форме прямоугольника и располагается в верхней части своей линии жизни. Сама линия жизни изображается вертикальной пунктирной линией, идущей от

центра прямоугольника, обозначающего объект. Внутри прямоугольника содержится информация, идентифицирующая линию жизни. Обычно это имя объекта, которое пишется посередине с маленькой буквы или имя класса, которое указывается после двоеточия. При этом вся запись может подчеркиваться, что является признаком объекта, который представляет собой экземпляр класса.

Следует помнить, что идентификатор линии жизни не может быть пустым, а сама линия жизни может быть ассоциирована только с одним объектом. В случае если отсутствует собственное имя объекта, должно быть указано имя класса и наоборот, если отсутствует имя класса, обязательно указывается имя объекта.

Объекты изображаются в верхней части диаграммы друг за другом. Крайним слева на диаграмме последовательности изображается объект, который начинает моделируемый процесс. Правее – следующий элемент системы, который непосредственно взаимодействует с первым и так далее. Таким образом, порядок расположения объектов на диаграмме последовательности определяется исключительно соображениями удобства визуализации их взаимодействия друг с другом.

Совсем не обязательно создавать все линии жизни одновременно. Отдельные линии жизни в системе могут создаваться по мере необходимости. В этом случае прямоугольник такой линии изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания экземпляра класса.

Отдельные линии жизни, выполнив свою роль в системе, могут быть уничтожены, чтобы освободить занимаемые ими ресурсы. Для обозначения момента уничтожения линии жизни в языке UML используется специальный символ в виде буквы «X». Ниже этого символа пунктирная линия не изображается, поскольку соответствующей линии жизни в системе уже нет, и она должна быть исключена из всех последующих взаимодействий. Если некоторая линия жизни была уничтожена, то вновь возникнуть в системе она уже не может. Вместо нее может быть создан другой экземпляр того же класса, который будет являться другой линией жизни.

4.4.2. Фокус управления

В процессе функционирования объектно-ориентированных систем объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия, или в состоянии пассивного ожидания сообщений от других объектов. Для того чтобы явно выделить подобную активность объектов, на диаграммах последовательности используется фокус управления.

Фокус управления (focus of control) **–** специальный символ на диаграмме последовательности, указывающий период времени, в течение которого объект выполняет некоторое действие, находясь в активном состоянии.

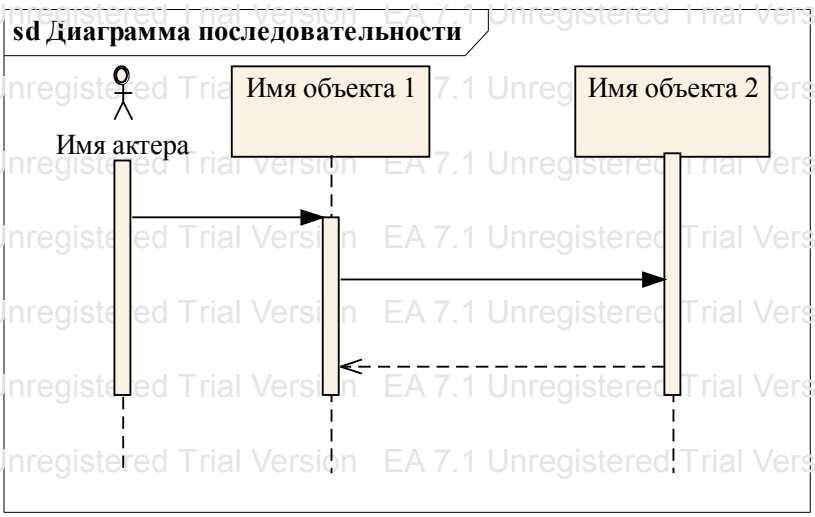


Рис. 4.12. Графическое изображение актера и объекта, которые активны на всем протяжении линии жизни.

Фокус управления изображается в форме вытянутого узкого прямоугольника, расположенного ниже соответствующего обозначения объекта. Верхняя сторона данного прямоугольника обозначает начало активности, а нижняя – окончание активности объекта.

Периоды активности объекта могут чередоваться с периодами пассивности. В этом случае фокусы управления меняют свое изображение на линию жизни и наоборот (рис. 4.11). Если объект активен на всем протяжении своей линии жизни, то он получает фокус управления сразу при своем создании. В этом случае, прямоугольник фокуса управления может заменять линию жизни объекта (рис. 4.12).

Следует заметить, что многие CASE-средства рисуют фокус управления автоматически, и разработчику не нужно заботиться о его изображении.

Иногда инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева в виде специального символа «проволочного человечка» со своим фокусом управления. Актер может иметь собственное имя или быть анонимным. Чаще всего, актер и его фокус управления существуют в системе постоянно, отмечая тем самым характерную для пользователя активность в инициировании взаимодействия с системой.

4.4.3. Сообщения

Реализация взаимодействий между элементами системы моделируется посредством сообщений, которые передаются между различными линиями жизни. Сообщения изображаются в виде стрелок различной формы и образуют некоторый порядок относительно времени своей передачи. При этом

сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже. Порядок наступления событий вдоль линии жизни существенен для обозначения последовательности, в которой эти события происходят. Однако расстояния между наступлениями событий на линиях жизни не имеют семантики.

На диаграммах последовательности чаще всего используются следующие типы сообщений:

- синхронное сообщение;
- асинхронное сообщение;
- ответное сообщение
- сообщение создания объекта.

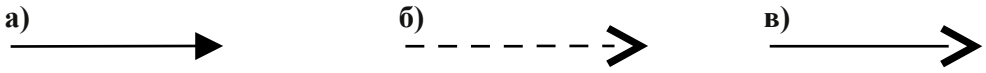


Рис. 4.13. Графическое изображение различных типов сообщений между объектами

Синхронное сообщение (synchCall) – используется для вызова методов, выполнения операций или обозначения отдельных вложенных потоков управления.

Эта наиболее распространенная разновидность сообщения изображается с помощью сплошной линии с закрашенной стрелкой на конце (рис. 4.13 а). Начинается эта линия от фокуса управления того объекта, который инициирует это сообщение, а заканчивается на линии жизни объекта, принимающего это сообщение и выполняющего в ответ определенные действия. При этом объект, который принимает сообщение, может получить фокус управления и стать активным, а объект, отправивший это сообщение, может потерять фокус управления и стать пассивным.

Особенностью данного вида сообщений является то, что синхронные сообщения приостанавливают поток управления до тех пор, пока не будет получен ответ.

Ответное сообщение (reply message) – используется для возврата из вызова процедуры или метода.

Графически ответное сообщение изображается пунктирной линией с открытой стрелкой на конце (рис. 4.13 б), которая направлена в сторону, противоположную тому сообщению, на которое необходимо ответить.

Такой же стрелкой изображается и сообщение *создания объекта (object creation)*. В этом случае стрелка сообщения указывает на созданный объект.

Следует отметить, что ответное сообщение не всегда изображается на диаграммах последовательности. В некоторых случаях оно может быть опущено, поскольку считается, что каждый вызов процедуры имеет свою пару – возврат вызова.

Асинхронное сообщение (asynchCall) – это сообщение, которое передается в произвольный момент времени и соответствует асинхронному вызову операции.

Передача такого сообщения обычно не сопровождается получением фокуса управления тем объектом, который принял данное сообщение. Асинхронные сообщения не ждут ответа и не приостанавливают поток управления, то есть сразу после отправления такого сообщения, происходит переход к следующему шагу, и последовательность взаимодействий продолжается.

На диаграммах последовательности этот вид сообщений изображается сплошной линией с открытой стрелкой на конце (рис. 4.13 в).

Важно отметить, что сообщение, моделирующее вызов операции и начало ее выполнения, ассоциируется с этой операцией, следовательно, данное сообщение имеет такое же имя, как и выполняемая операция того класса, к которому сообщение направлено. При этом сообщение может содержать также список аргументов, который должен соответствовать той операции класса, на которую ссылается это сообщение.

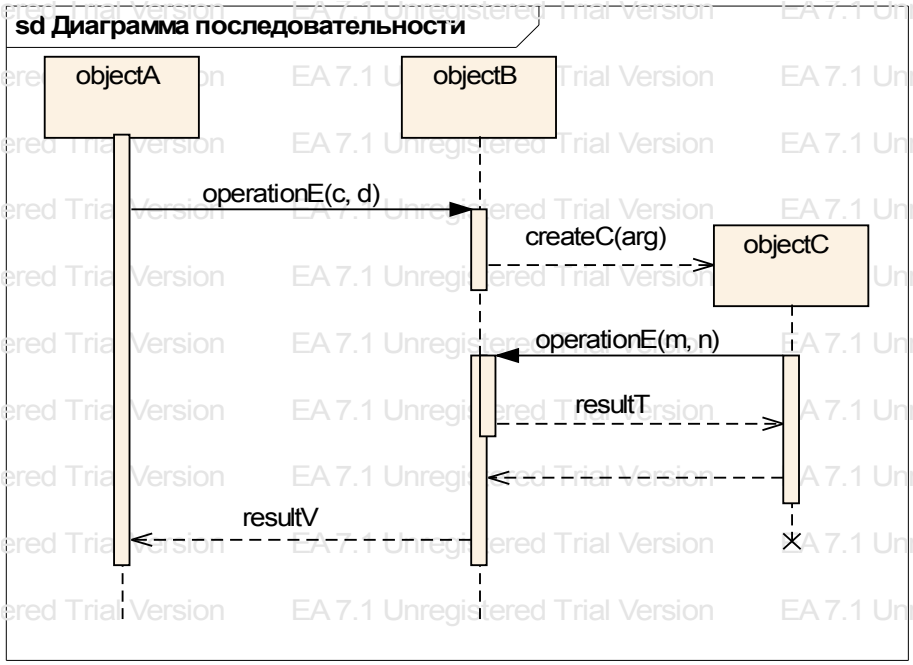


Рис. 4.14. Изображение сообщений с именем и списком аргументов

Таким образом, проектирование диаграммы последовательности затруднительно без использования диаграммы классов и диаграммы вариантов использования.

Из диаграммы вариантов использования определяются действующие лица и вариации поведения, описанные в сценарии к вариантам использования, а из диаграммы классов определяются операции, осуществляемые над объектами, участвующими во взаимодействии.

В свою очередь диаграмма последовательности позволяет уточнять и дорабатывать диаграмму классов следующим образом: если в ходе разработки диаграммы последовательности операций, определенных для некоторого класса, оказалось недостаточно для спецификации всех рассматриваемых взаимодействий, то эти операции должны быть добавлены к соответствующему классу.

Таким образом, проектирование диаграммы последовательности следует начинать после построения диаграммы классов и диаграммы вариантов использования, так как обе эти диаграммы непосредственно используются при моделировании соответствующей последовательности взаимодействий.

4.5. Моделирование альтернативных потоков управления

Как уже говорилось, диаграммы последовательности должны быть спроектированы не только для основного потока событий, но и для альтернативного. Можно строить отдельную диаграмму последовательности для каждого исключения, а можно использовать средства языка UML позволяющие наглядно визуализировать подобные процессы на общей диаграмме.

Одним из таких средств является *ветвление потока управления*. Для изображения ветвления используются две или более стрелки, выходящие из одной точки фокуса управления объекта. При этом рядом с каждой из них должно быть явно указано соответствующее условие ветви в форме булевского выражения, которое записывается в квадратных скобках. Запись этих условий должна исключать одновременную передачу альтернативных сообщений по двум или более ветвям. В противном случае на диаграмме последовательности может возникнуть конфликт ветвления, и такая диаграмма не будет считаться согласованной.

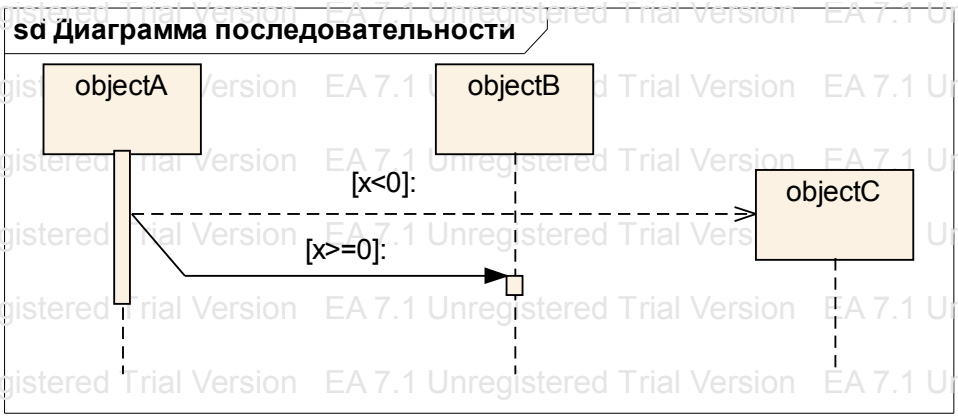


Рис. 4.15. Изображение ветвления потока управления

Количество ветвей может быть произвольным, однако наличие ветвлений может существенно усложнить интерпретацию диаграммы последовательности, поэтому ветвление – конструкция для диаграмм последовательности непопулярная и используется в них очень редко. В основном для представления альтернативных потоков управления на диаграммах последовательности используются комбинированные фрагменты взаимодействия.

4.5.1. Комбинированный фрагмент взаимодействия

Комбинированный фрагмент взаимодействия (combined fragment) – это мощный механизм моделирования вложенного поведения, с помощью которого разработчик может представить несколько траекторий взаимодействий в компактной форме.

Комбинированный фрагмент определяется посредством *оператора взаимодействия*, который определяет тип комбинированного фрагмента взаимодействия, и соответствующих ему *операндов взаимодействия*, предназначенных для изображения внутренней части комбинированного фрагмента.

Операнды взаимодействия совместно образуют фрейм (прямоугольник) комбинированного фрагмента, внутри которого они отделяются друг от друга пунктирной линией.

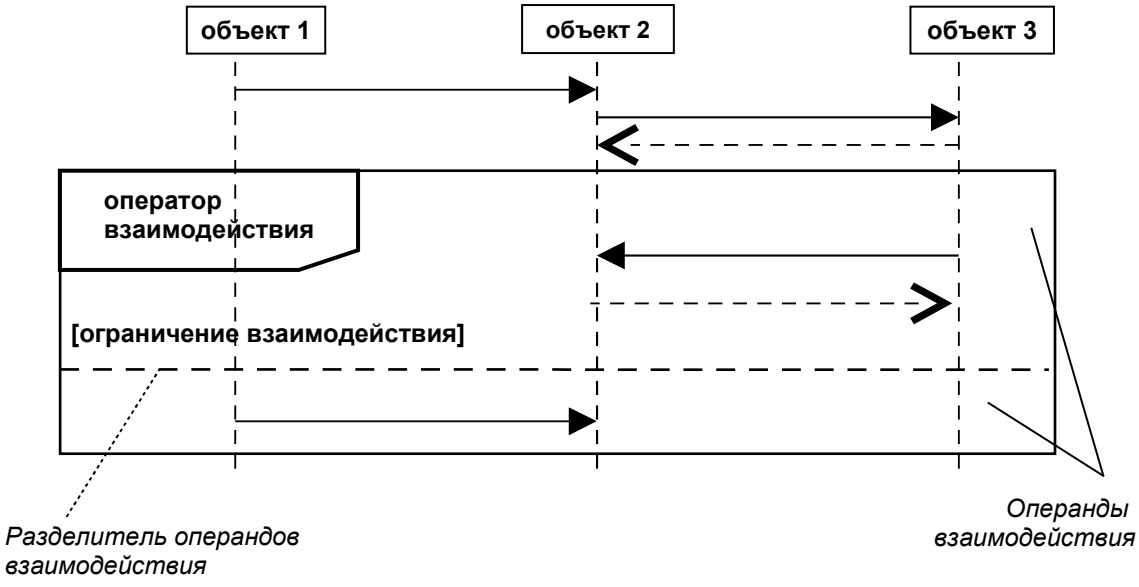


Рис. 4.16. Графическое изображение комбинированного фрагмента.

Операнд взаимодействия может иметь ***ограничение взаимодействия (interaction constraint)*** - это логическое выражение, которое выступает в роли сторожевого условия выполнения взаимодействия.

Графически оператор взаимодействия обозначается в виде прямоугольника с именем оператора взаимодействия внутри пятиугольника в левом верхнем углу.

Ограничение взаимодействия изображается в квадратных скобках в соответствующем фрагменте взаимодействия и определяет условия выполнения соответствующих сообщений.

Всего таких операторов, определяющих тип комбинированного фрагмента, двенадцать: *alt, assert, break, critical, ignore, consider, loop, neg, opt, par, seq, strict.* Рассмотрим семантику наиболее распространенных из этих операторов.

4.5.2. Оператор взаимодействия *break*

Оператор взаимодействия ***break (завершение)***, определяет комбинированный фрагмент, который представляет некоторый сценарий завершения. Этот сценарий может выполняться вместо оставшейся части диаграммы последовательности, которая содержит этот оператор взаимодействия, поэтому оператор завершения должен охватывать все линии жизни данной диаграммы взаимодействия.

Оператор *break* не может содержать более одного операнда, у которого обязательно должно быть сторожевое условие. Если это условие выполняется, то выполняется и комбинированный фрагмент, а оставшаяся часть диаграммы будет пропущена. Если сторожевое условие не выполняется, то игнорируется сам оператор взаимодействия, а следующим будет выполняться сообщение идущее сразу за прямоугольником комбинированного фрагмента.

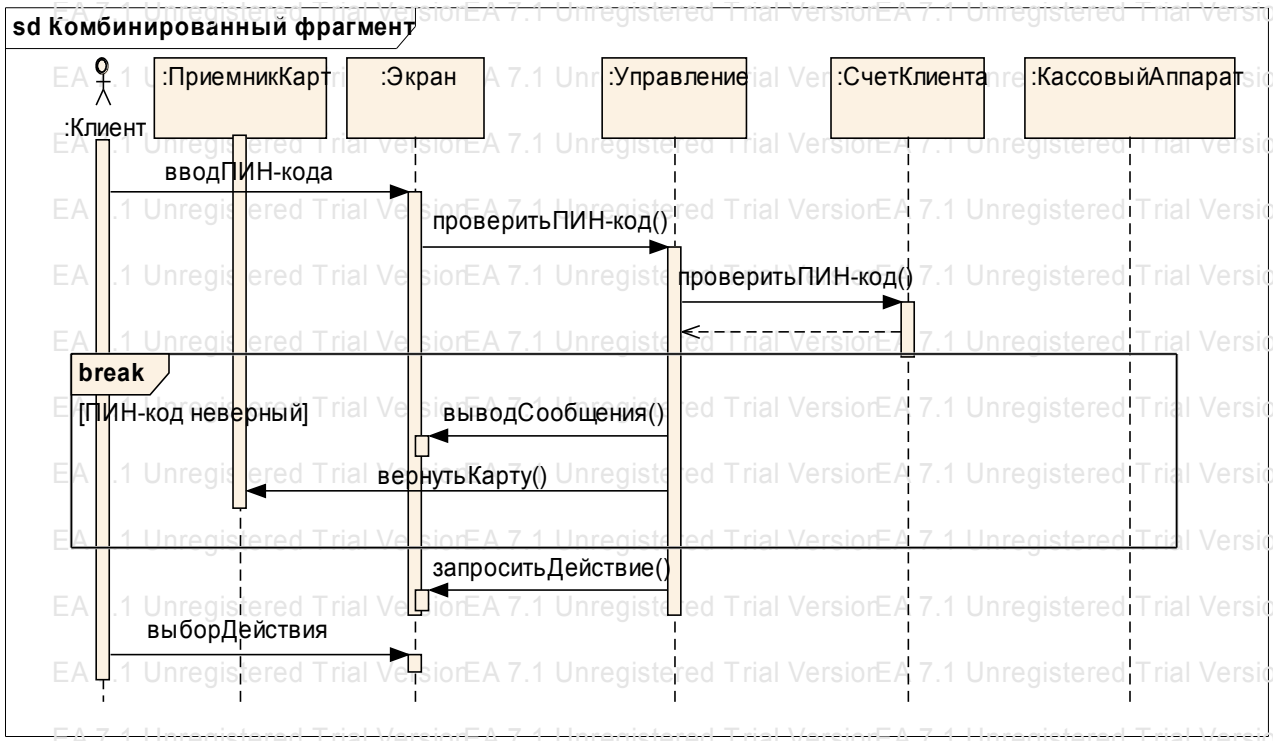


Рис. 4.17. Пример комбинированного фрагмента *break*.

На рис. 4.17 представлена часть диаграммы последовательности с комбинированным фрагментом завершения. При вводе неверного ПИН-кода поток управления передается в комбинированный фрагмент, внутри которого выполняются сообщения «выводСообщения» и «вернутьКарту» и на этом выполнение последовательности взаимодействий завершается, то есть все сообщения, расположенные ниже комбинированного фрагмента завершения игнорируются. Если ПИН-код оказывается правильным, то сторожевое условие внутри комбинированного фрагмента не выполняется. Это значит, что сам комбинированный фрагмент пропускается, а выполнение последовательности взаимодействий продолжается сообщением «запроситьДействие», идущим сразу после комбинированного фрагмента.

4.5.3. Оператор взаимодействия *loop*

Оператор взаимодействия ***loop (цикл)*** представляет собой циклическое повторение некоторой последовательности сообщений, то есть операнд цикла может повторяться несколько раз.

Оператор *loop* содержит дополнительное сторожевое условие, которое может включать некоторое логическое выражение, а также нижний и верхний пределы числа повторений цикла.

Число повторений указывается в круглых скобках после оператора *loop*, сначала нижний, а через запятую верхний предел повторений.

Если в данном операторе взаимодействия определено только одно значение, то считается, что минимальное количество повторений равно максимальному количеству повторений.

Если указан только оператор *loop*, то это означает цикл с бесконечной верхней границей и с 0 в качестве нижней границы.

Количество повторений цикла не может быть меньше нижнего предела повторений и больше верхнего.

После того как минимальное число повторений будет выполнено, проверяется логическое условие в квадратных скобках, то есть ограничение. Если оно не выполняется, то цикл на этом заканчивается. Если же логическое выражение принимает значение «истина», то происходит еще одно выполнение цикла, если количество итераций не превысило верхнего предела повторений. После этого снова проверяется логическое условие и так до тех пор, пока либо логическое выражение не примет значение «ложь», либо количество повторений не достигнет верхнего предела.

На рис. 4.18 рассмотрен несколько измененный фрагмент диаграммы последовательности из предыдущего примера. В этом случае возможен повторный ввод пин-кода, если код был введен с ошибкой. Цикл будет выполнен, по крайней мере, один раз, так как при первом проходе сторожевое условие не проверяется. Цикл завершается после первого успешного ввода пин-кода или после трех неудачных попыток.

В зависимости от условий завершения комбинированного фрагмента *loop,* последовательность взаимодействий будет либо прервана в операторе завершения, где карточка клиента будет блокирована после трех неудачных попыток ввода ПИН-кода, либо продолжена дальше в том случае, если хотя бы одна из трех попыток оказалась успешной.

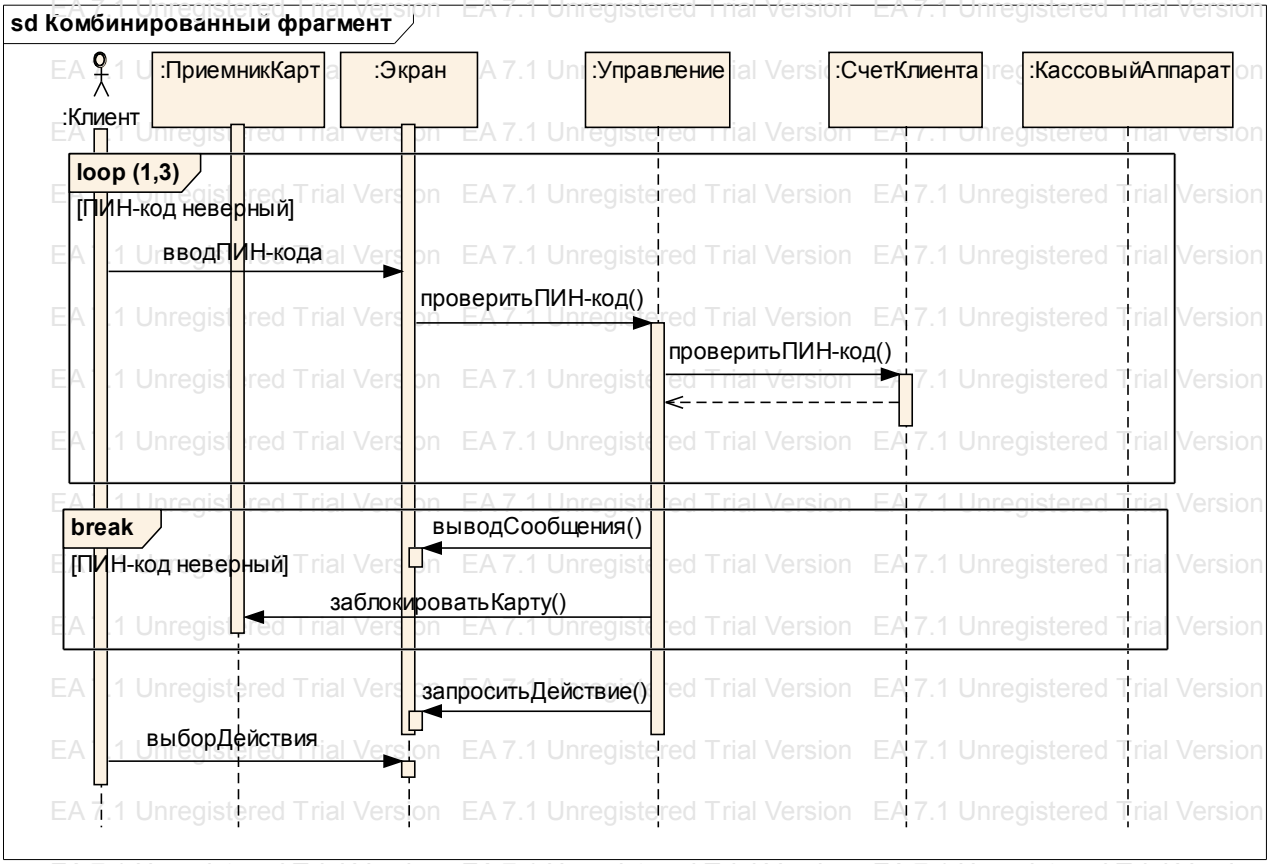


Рис. 4.18. Пример комбинированного фрагмента *loop*.

4.5.4. Оператор взаимодействия *alt*

Оператор взаимодействия ***alt (альтернатива)*** специфицирует комбинированный фрагмент, предоставляющий некоторый выбор поведения минимум из двух операндов. Каждый из операндов должен иметь сторожевое условие его выполнения, если такого условия нет, то неявно предполагается значение «истина». При этом выбор может быть сделан не более одного из операндов, то есть несколько ограничений из разных операндов не могут одновременно принимать значение «истина».

Если при выполнении последовательности взаимодействий, ни одно из сторожевых условий, соответствующих операндам данного комбинированного фрагмента, не приняло значение «истина», то не выполняется ни один из операндов этого комбинированного фрагмента. Фрагмент пропускается полностью и выполняется следующая за ним часть диаграммы взаимодействия.

Оператор взаимодействия *alt* может иметь операнд, помеченный сторожевым условием [else]. Этот операнд будет выполнен в том случае, если

ни одно из сторожевых условий остальных операндов данного комбинированного фрагмента не приняло значения «истина».

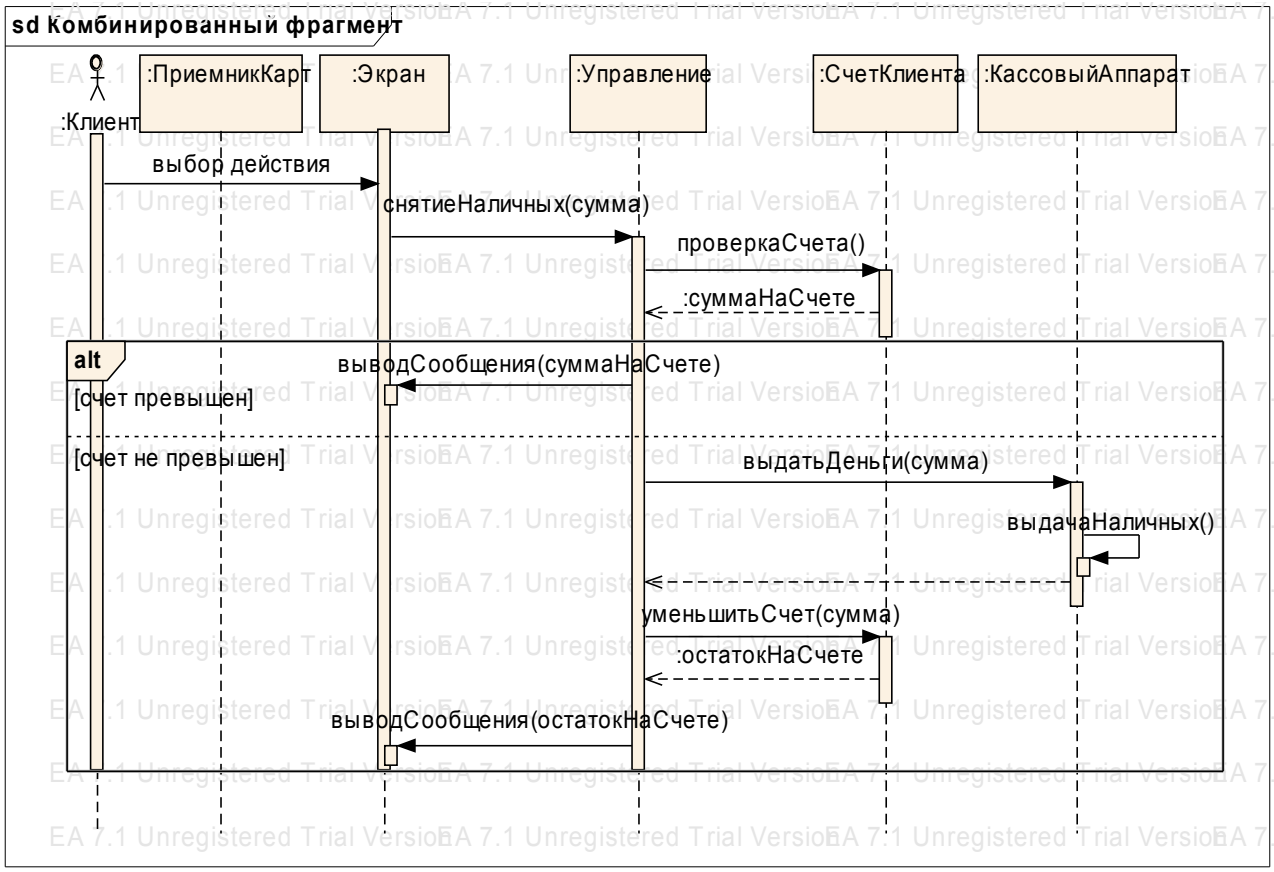


Рис. 4.19. Пример комбинированного фрагмента *alt*.

В качестве примера рассмотрен упрощенный вариант диаграммы последовательности, которая описывает снятие денег в банкомате (рис. 4.19). Клиент запрашивает для снятия определенную сумму, если данная сумма превышает количество денег на счете, то выполняется первый операнд, а второй операнд будет пропущен. Если сторожевое условие «счет превышен» не выполняется, то первый операнд будет пропущен, после чего проверяется условие второго операнда и сам этот операнд выполняется.

4.5.5. Оператор взаимодействия *opt*

Частным случаем оператора *alt* можно назвать оператор взаимодействия ***opt (необязательный)***, так как он семантически эквивалентен альтернативному комбинированному фрагменту, в котором имеется один операнд с непустым содержанием, а второй операнд отсутствует. Оператор *opt* специфицирует комбинированный фрагмент поведения, который представляет собой выбор поведения, когда выполняется один операнд или вовсе ничего не выполняется. Необязательный комбинированный фрагмент состоит из одного операнда со

сторожевым условием. Если оно выполняется - выполняется и операнд. В противном случае операнд не выполняется.

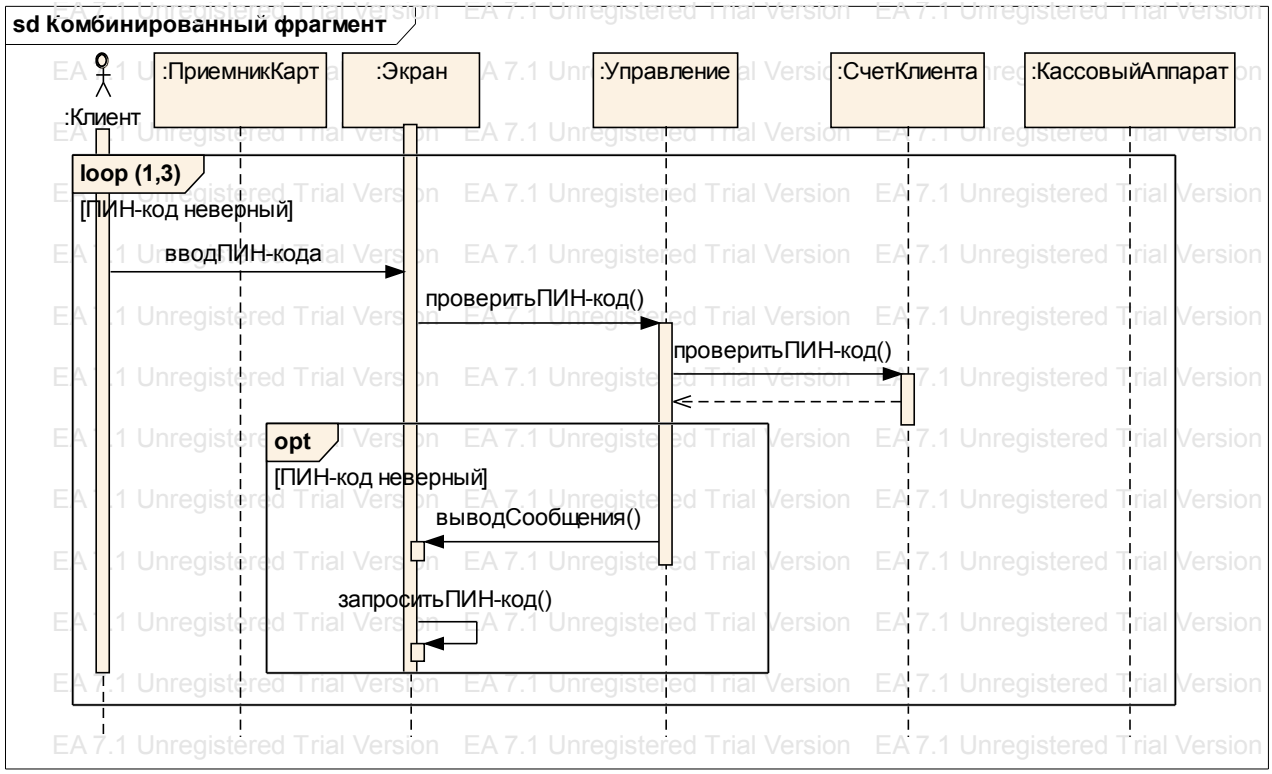


Рис. 4.20. Пример комбинированного фрагмента *opt*.

На рис. 4.20 рассмотрен пример фрагмента диаграммы последовательности для ввода ПИН-кода. В данном случае комбинированный фрагмент *opt* используется внутри оператора взаимодействия *loop* для необязательного вывода сообщения и запроса ПИН-кода в случае его неверного ввода, если ПИН-код окажется верным, то оператор *opt* будет проигнорирован.

Следует обратить внимание, что комбинированный фрагмент (кроме оператора break) может охватывать не все линии жизни, имеющиеся на диаграмме последовательности, а только те, которые участвуют во взаимодействии внутри данного фрагмента. Это правило не является обязательным и используется только для удобства визуализации.

В приведенном примере на рис. 4.19 оператор взаимодействия *opt* расположен на тех линиях жизни, которые участвуют в передаче сообщений при выполнении данного оператора.

4.5.6. Оператор взаимодействия *par*

Наряду с последовательными сообщениями на диаграммах последовательности можно отображать и параллельное выполнение взаимодействий. Для этого используется оператор взаимодействия ***par (параллельный)***, который специфицирует комбинированный фрагмент,

представляющий некоторое параллельное выполнений взаимодействий своих операндов.

В параллельном комбинированном фрагменте наступление событий у различных операндов могут чередоваться по времени произвольным образом. Однако внутри каждого отдельного операнда соблюдается последовательный порядок выполнения сообщений.

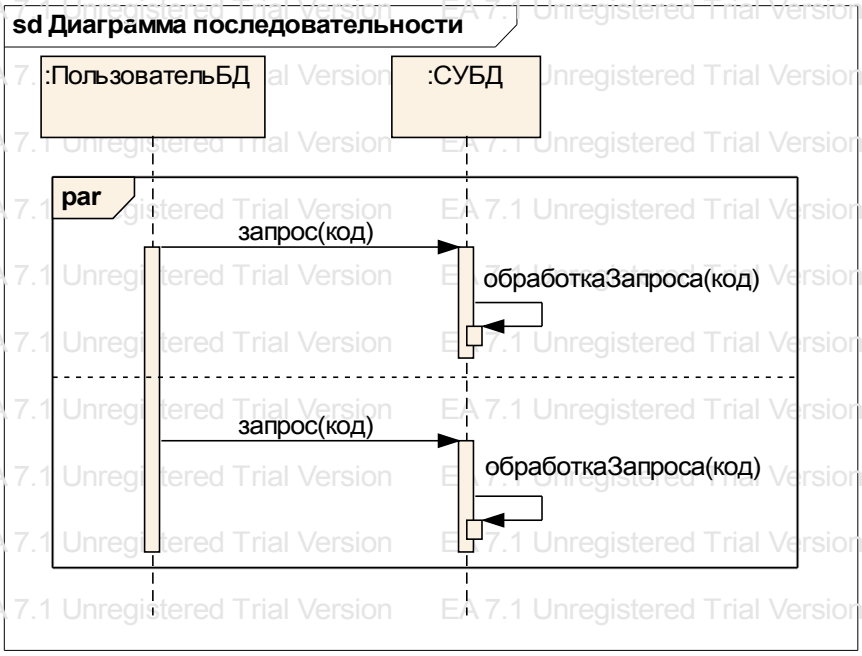


Рис. 4.21. Пример комбинированного фрагмента par.

На рис. 4.21 приведен фрагмент диаграммы последовательности, которая моделирует обработку запросов некоторой СУБД. В этом примере все запросы могут обрабатываться параллельно, причем запросы от разных пользователей и их обработка могут чередоваться между собой. Например, после передачи запроса в первом операнде может быть передан запрос во втором операнде. После чего обработан запрос в первом операнде, и, наконец, обработан запрос во втором операнде. То есть из расположения сообщений в различных регионах операндов нельзя сделать никакого заключения о порядке их выполнения. В то же время сообщения внутри каждого операнда упорядочены сверху вниз.

Совместно с использованием оператора параллельности на практике часто используется оператор взаимодействия *critical (критический регион)*.

4.5.7. Оператор взаимодействия *critical*

Оператор взаимодействия ***critical (критический регион)*** специфицирует комбинированный фрагмент взаимодействия, траектории которого не могут чередоваться с другими спецификациями, наступления событий на тех или иных линиях жизни, которые этот регион покрывает.

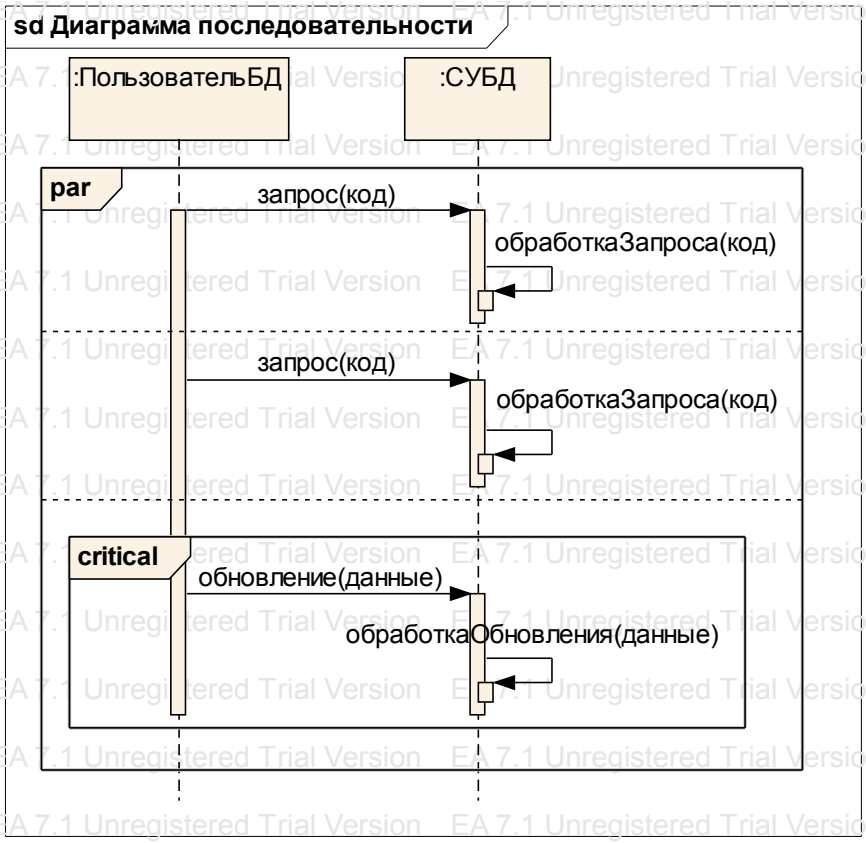


Рис. 4.22. Пример комбинированного фрагмента *critical.*

Это означает, что наступления событий или передача сообщений в этом регионе не может чередоваться с наступлениями событий или передачами сообщений в других регионах, даже если другие операторы взаимодействия могут допускать такое чередование.

В качестве иллюстрации применения *критического региона* рассмотрим фрагмент диаграммы последовательности из предыдущего примера. Добавление на диаграмму критического региона означает, что если поступает запрос на обновление, который изображен в комбинированном фрагменте «critical», то он должен быть сразу обработан. При этом между поступлением запроса на обновление и его обработкой СУБД не могут быть выполнены никакие другие сообщения.

Использование рассмотренных выше способов моделирования вложенного поведения позволяет изобразить несложные случаи ветвления потока управления на одной диаграмме последовательности. Однако, при проектировании более сложных случаев, для описания отдельной ветви управления может потребоваться отдельная диаграмма последовательности, так как каждый альтернативный поток затрудняет понимание построенной модели.

Итак, мы рассмотрели основные элементы диаграмм последовательности, которые в графической форме прорабатывают и детализируют неформальное описание системы, заданное вариантами использования и наглядно показывают всех участников взаимодействия.

Таким образом, моделирование диаграмм последовательности позволяет более детально показать поведение системы в процессе выполнения соответствующего варианта использования и на этом основании уточнить и дополнить разработанную ранее диаграмму классов.

4.6. Диаграммы деятельности

При разработке модели поведения проектируемой системы необходимо не только представить процесс изменения ее состояний, но и отразить особенности реализации выполняемых системой операций. Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности (активности).

Диаграммы деятельности (activity diagram) – служат для моделирования последовательности действий, которые выполняются различными элементами, входящими в состав системы.

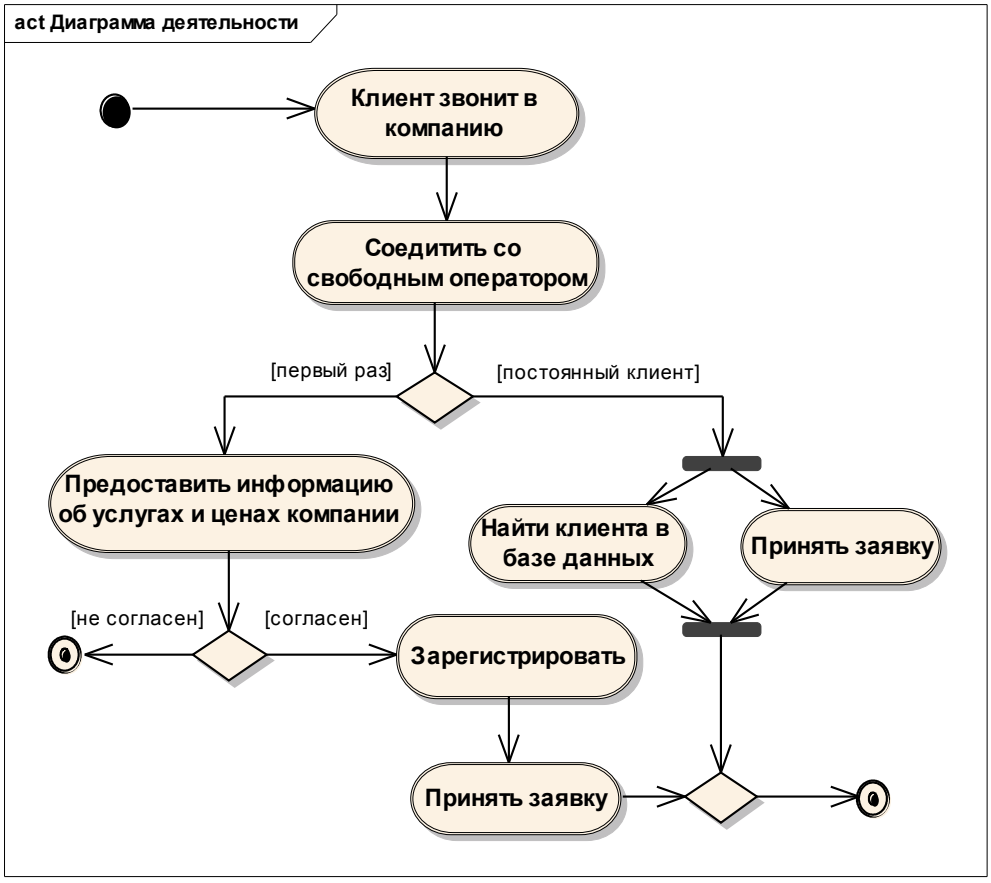


Рис. 4.23. Пример диаграммы деятельности.

Другими словами, этот вид диаграмм раскрывает детали алгоритмической реализации операций, выполняемых системой, поэтому диаграмма деятельности похожа на обычную блок-схему. Однако, в отличие от блок-схемы, диаграмма деятельности может показывать одновременно

параллельную и последовательную деятельность, поэтому диаграммы активности удобно использовать для моделирования систем, состоящих из множества объектов, одновременно выполняющих множество операций.

Диаграммы деятельности или активности предназначены для уточнения вариантов использования и моделей последовательности. Диаграммы активности показывают поток управления, подобно диаграммам последовательности, но сосредотачивают внимание на операциях, а не на объектах. В общем случае, этот тип диаграмм может относиться к отдельному варианту использования, пакету, классу, операции класса или системе в целом, то есть диаграммой деятельности можно дополнить любой элемент модели, имеющий динамическое поведение.

Диаграммы деятельности являются частным случаем диаграммы состояний, так как элементами диаграмм активности являются операции, а именно виды деятельности из модели состояний. Однако диаграммы деятельности имеют свои особенности.

В общем случае деятельность является самостоятельным элементом поведения, которая в свою очередь может включать в себя другие деятельности или отдельные действия.

Действие (action) – это элементарная единица поведения, а ***деятельность (activity)*** состоит из последовательности выполнения действий. Эта последовательность действий описывается с помощью потока управления, который лежит в основе понимания модели поведения в представлении диаграммы деятельности.

Поток управления моделируется в форме узлов деятельности, соединенных дугами деятельности, которые являются основными элементами диаграммы.

4.6.1. Узлы и дуги деятельности

Семантически ***узел деятельности (activity)*** - является элементом диаграммы, который служит для представления процедурной последовательности действий, требующих определенного времени, а ***узел действия (action)*** моделирует шаг выполнения алгоритма или процедуры.

Графически узел деятельности и узел действия изображаются одинаковой фигурой - прямоугольником со скругленными углами, внутри которого пишется имя действия (рис. 4.23).

Имя действия может быть записано на естественном языке или языке программирования. Никаких дополнительных ограничений при записи действий не накладывается.

Узлы деятельности могут быть связаны между собой переходами или ***дугами деятельности (control flow)***, каждая из которых является направленным отношением и имеет в качестве источника и цели некоторые узлы. Дуга деятельности изображается сплошной линией со стрелкой. Такая

линия без надписи, соединяющая две деятельности на диаграмме, означает, что последующая деятельность начинается сразу, как только закончиться предыдущая деятельность, поэтому если из узла действия выходит единственная дуга, то ее можно никак не помечать.

Если же таких дуг несколько, то при моделировании последовательной деятельности переход к следующему действию может осуществляться только по одной из них. В этом случае для каждой из таких дуг должно быть явно записано соответствующее сторожевое условие в квадратных скобках радом с символом перехода. При этом для всех выходящих из некоторого узла действия дуг должно выполняться требование истинности только для одной из них.

Такая ситуация возникает тогда, когда последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения промежуточного результата. Ветвление на диаграмме деятельности моделируются с помощью специальных символов, которые называются узлы управления.

4.6.2. Узлы управления

Узлы управления используются на диаграммах деятельности для координации потока управления. Существует несколько разновидностей узлов управления:

- начальный узел *(initial)*
- финальный узел *(final)*
- узел финала потока *(flow final)*
- узел соединения *(merge)*
- узел решения *(decision)*
- узел разделения *(fork)*
- узел слияния *(join)*.

Рассмотрим назначение и нотацию каждого из этих узлов.

Каждая диаграмма деятельности должна иметь начальный и конечный узлы управления.

Начальный узел (initial) является узлом управления, в котором начинается поток при вызове деятельности, то есть это начальная точка выполнения деятельности. Начальный узел обозначается сплошным кружком со стрелкой (рис. 4.23).

Узел финала деятельности (final) является узлом управления, который прекращает или останавливает все потоки в деятельности. Узел финала деятельности обозначается символом «бычий глаз» и не может иметь исходящих стрелок.

Деятельность может иметь более чем один узел финала деятельности. При достижении первого из них прекращаются все потоки в этой деятельности, и сама деятельность завершается (рис. 4.23).

Если по какой-либо причине нежелательно завершать все потоки в деятельности, то вместо узлов финала деятельности можно использовать узлы финала потока.

Узел финала потока (flow final) является узлом управления, который завершает отдельный поток управления, не завершая содержащей его деятельности. От финального узла он отличается тем, что узел финала потока прекращает выполнение только одного потока управления и не оказывает влияния на другие потоки этой деятельности, то есть деятельность не завершается и может выполняться в других потоках. Попадание же в любой финальный узел приводит к завершению всех потоков управления, то есть вся деятельность прекращается.

Обозначается конечное состояние потока в виде окружности с крестом.

Узел решения (decision) на диаграмме деятельности является узлом управления, который выбирает между выходящими потоками.

Графически узел решения изображается в виде ромба, внутри которого нет никакого текста (рис. 4.24). Он должен иметь дугу деятельности, входящую в него и одну или несколько дуг, выходящих из него. Каждая деятельность, достигнув узла решения, может перейти только по одной выходящей дуге. Если деятельность имеет несколько последующих элементов, каждая из выходных дуг должна иметь сторожевое условие, которое возвращает значение «истина» или «ложь». При оценивании всех сторожевых условий узла решения только одно может принимать значение «истина». Если условие оказывается истинным, то соответствующая ему дуга указывает на деятельность, подлежащую выполнению.

Если не выполняется ни одно из условий, то модель считается плохо согласованной, так как система в этом случае зависнет. Во избежание подобных ситуаций следует использовать ключевое слово [*else*]. Это условие является допустимым для деятельности, которая не принята ни одной другой дугой, выходящей из узла решения.

Если для узла решения оказываются выполненными несколько сторожевых условий, то будет запущена только одна деятельность, причем невозможно определить какая. В этом случае, модель так же будет являться некорректной, поэтому разработчик должен обязательно учитывать возможности перекрытия сторожевых условий для узла решения.

Для объединения нескольких альтернативных ветвей на диаграмме деятельности используется аналогичный символ в виде ромба, который в этом случае называют *узел соединения* (рис. 4.24).

Узел соединения (merge) является узлом управления, который объединяет вместе несколько альтернативных потоков.

Входящих дуг у символа соединения может быть несколько, а выходить из него может только одна дуга. При этом выходящая дуга не должна содержать сторожевых условий. Использование этого символа упрощает визуальный контроль логики выполнения процедурных действий на диаграмме деятельности.

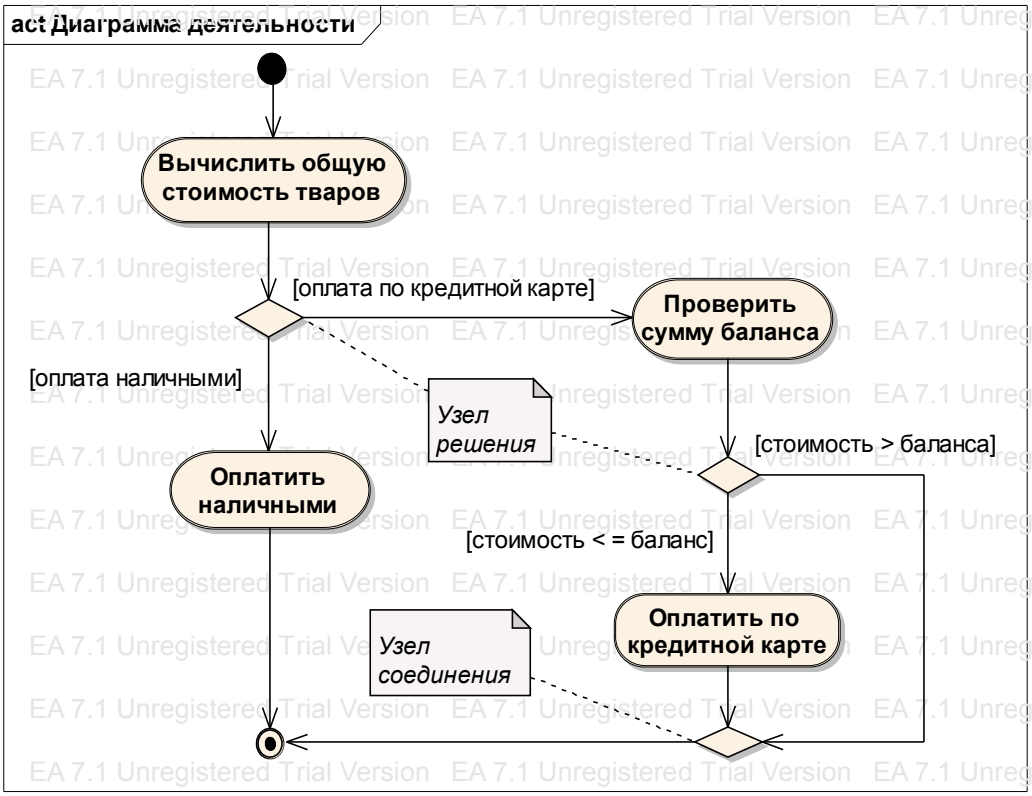


Рис. 4.24. Изображение ветвления на диаграмме деятельности.

Как уже говорилось, в отличие от блок-схемы диаграмма деятельности может показывать одновременно параллельную и последовательную деятельности.

Для описания параллельной деятельности в языке UML используются специальный символ узла управления, который используется для визуализации разделения и слияния параллельных вычислений или потоков управления.

Рис. 4.25. Графическое изображение разделения и слияния параллельных потоков управления.

Узел разделения (fork) является узлом управления, который расщепляет поток на несколько параллельных потоков.

Узел разделения обозначается сплошной толстой линией и имеет одну входящую дугу и несколько исходящих, которые соответствуют параллельным видам деятельности. Параллельность означает допустимость произвольного порядка выполнения и завершения всех видов деятельности.

Узел слияния (join) является узлом управления, который синхронизирует несколько параллельных потоков.

Узел слияния обозначается аналогично узлу разделения сплошной толстой линией. Он имеет несколько входящих дуг и только одну исходящую дугу. Узел слияния специфицирует слияние управления, то есть объединение нескольких параллельных потоков управления в один. Объединение возможно только после полного завершения всех видов параллельной деятельности входящих в узел слияния. Другими словами, для синхронизации необходимо, чтобы управление было передано в точку слияния по всем входящим видам деятельности.

В качестве примера, иллюстрирующего особенности изображения ветвления потоков управления, рассмотрим вариант диаграммы деятельности для регистрации пассажиров в аэропорту (рис. 4.26).

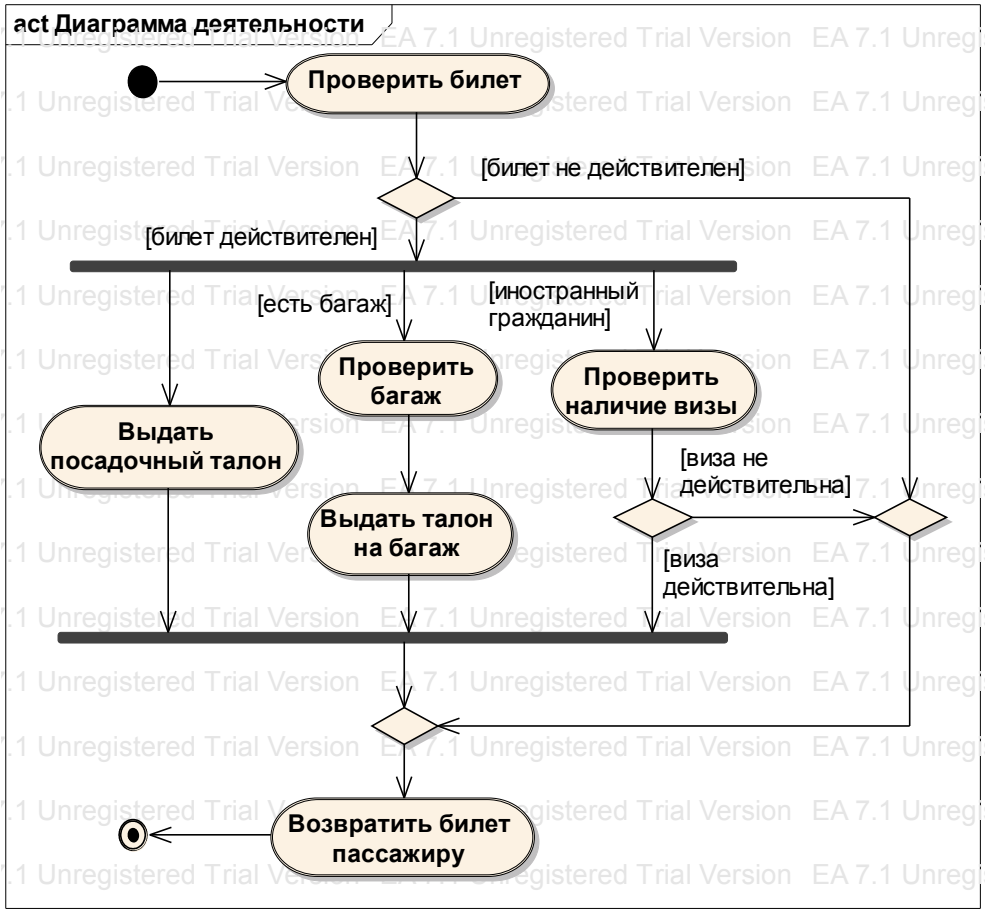


Рис. 4.26. Пример диаграммы деятельности для регистрации пассажиров.

Первоначально выполняется проверка билета. Если он оказывается не действительным, то билет просто возвращается пассажиру и никакой дополнительной деятельности при этом не происходит. Если же билет действителен, то поток управления передается по трем параллельным направлениям. Пассажиру выдается посадочный талон.

Параллельно проверяется наличие багажа и выдается талон на багаж. Если пассажир иностранный гражданин, то проверяется наличие визы. Если виза действительна, и завершены все описанные выше параллельные действия, то деятельность синхронизируется, то есть пассажиру возвращается билет, и он может следовать на посадку с билетом и посадочным талоном, а поток управления оформления пассажира переходит в финальный узел.

Если виза оказывается не действительной, то не выдается посадочный талон и не оформляется багаж, а просто возвращается билет, и поток управления переходит в финальный узел, то есть все действия сотрудников аэропорта прекращаются.

4.6.3. Составная деятельность

Деятельность, которая состоит из нескольких более простых действий, может быть представлена на диаграмме активности в виде ***составной деятельности (structured activity),*** которая обозначается специальной пиктограммой в нижнем правом углу символа деятельности. Этот символ означает, что подробное содержание данной деятельности вынесено на вложенную диаграмму активности.

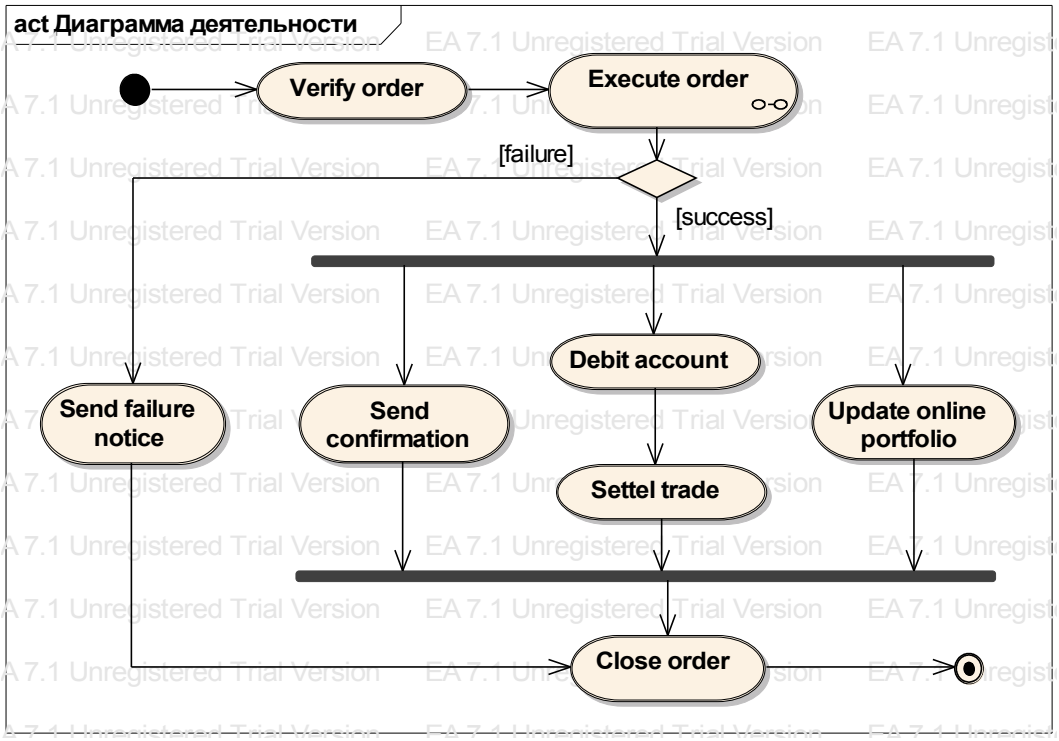


Рис. 4.27. Графическое изображение составной деятельности.

Иначе говоря, сложная деятельность на диаграммах активности может быть разложена на более мелкие составляющие. Однако следует помнить, что виды деятельности на одной диаграмме должны относиться к одному уровню детализации. Сохранить уровень детализации можно, если описать составляющие сложного действия на отдельной диаграмме активности.

В качестве примера изображения составной деятельности рассмотрим диаграмму активности для брокерской системы обработки заказов на покупку акций (рис. 4.27).

Сетевая брокерская система начинает с проверки заказа. Затем заказ реализуется на фондовой бирже. Если выполнение заказа происходит успешно, система выполняет три действия одновременно: отправляет клиенту подтверждение, обновляет сетевой портфель ценных бумаг с учетом результатов сделки и заканчивает сделку со второй стороной, снимая средства со счета клиента и перечисляя наличные или ценные бумаги. Когда все три параллельных потока завершаются, система закрывает заказ. Если же выполнение заказа оказывается неудачным, система отправляет сообщение об этом клиенту и закрывает заказ.

Все деятельности, представленные на данной диаграмме, относятся к одному уровню детализации. Если бы один из этих этапов был бы разбит на более мелкие составляющие, другие этапы тоже следовало бы разбить на мелкие шаги, чтобы все элементы диаграммы остались на одинаковом уровне. Такое деление существенно затруднит визуальное восприятие процесса выполнения деятельности на данной диаграмме, поэтому деятельность *«Execute order»* отмечена пиктограммой составной деятельности, а ее внутренняя структура отображена на дополнительной диаграмме активности (рис. 4.28).

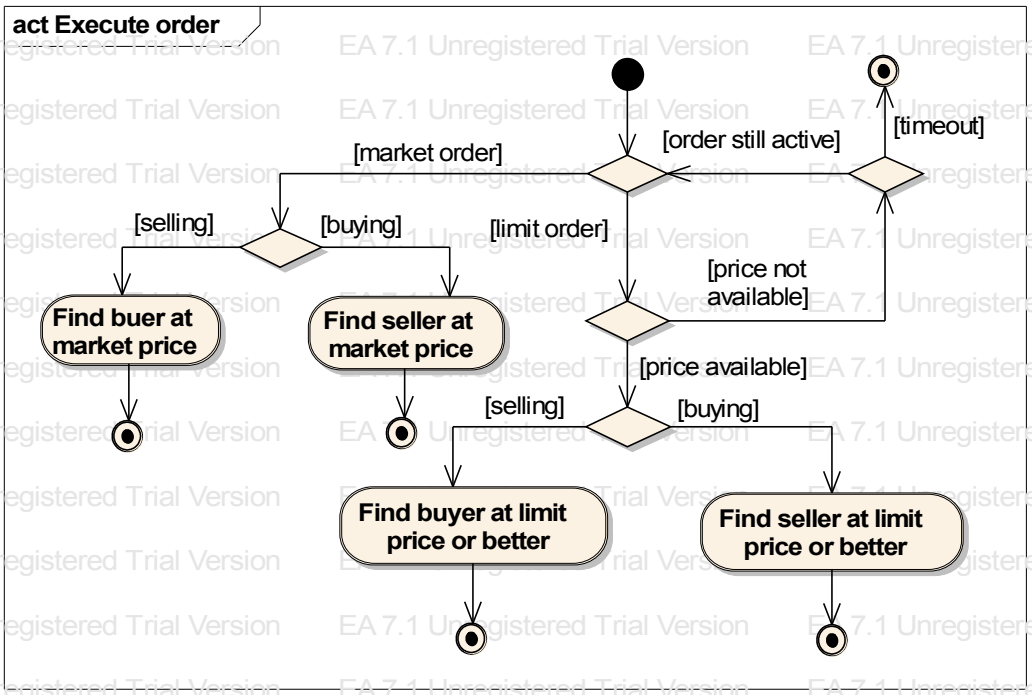


Рис. 4.28. Вложенная диаграмма деятельности для действия *«Execute order»*.

4.6.4. Разбиение деятельности

В моделях деятельности часто бывает удобно представить, какое именно организационное подразделение или экземпляр класса отвечает за ту или иную деятельность. Такой вариант представления диаграмм деятельности называется *разбиение деятельности.*

Разбиение деятельности (activity partition) – это элемент модели, предназначенный для группировки действий, которые относятся к одной деятельности и имеют некоторую общую характеристику

Графическое обозначение разбиения деятельности получило название ***«плавательные дорожки» (swimlanes).***

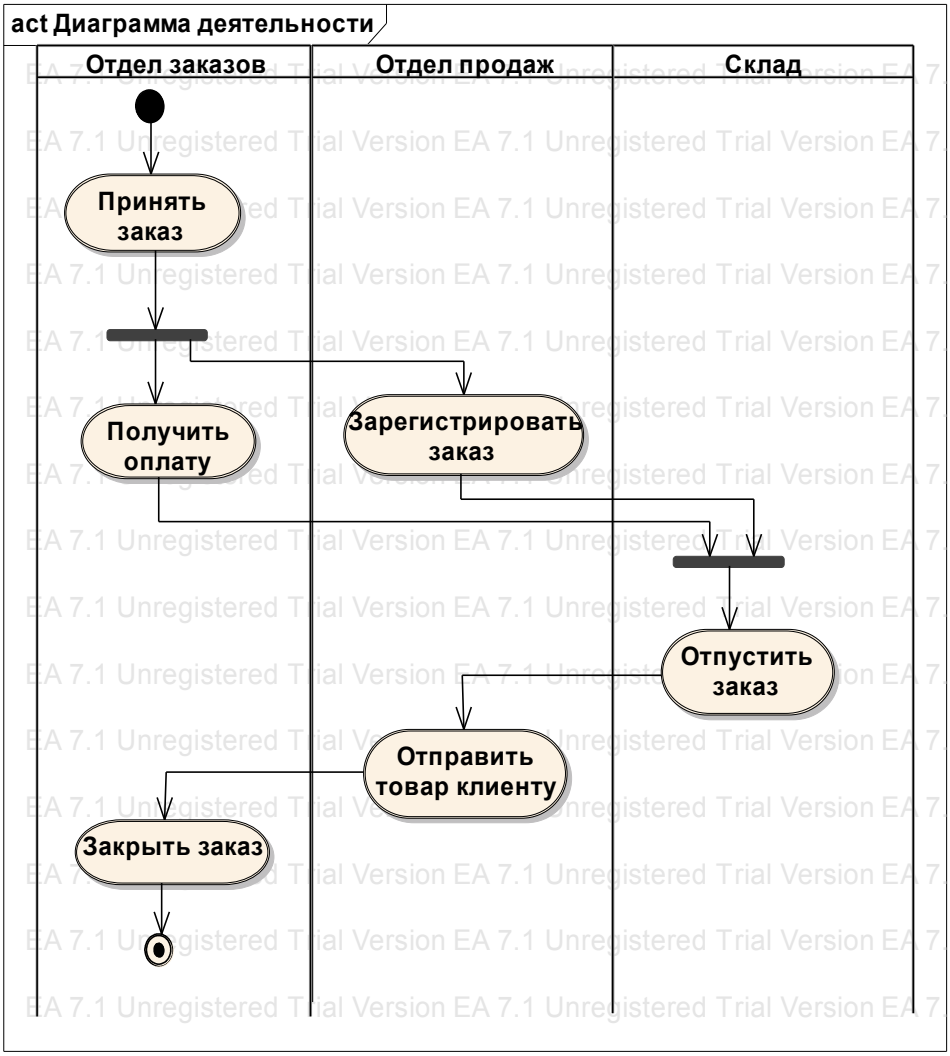


Рис. 4.29. Диаграмма деятельности с плавательными дорожками

Помещение деятельности внутрь плавательной дорожки означает, что она выполняется экземпляром того класса, которому соответствует данное разбиение. Названия подразделения или экземпляра класса явно указываются в верхней части разбиения деятельности. Пересекать линию, разделяющую дорожки, могут только дуги деятельности, которые в этом случае обозначают

переход потока управления от одного элемента системы к другому. Порядок следования дорожек не несет никакой семантической информации и выбирается из соображений удобства визуализации.

На рис. 4.29 приведен пример диаграммы активности с плавательными дорожками, которые визуализируют информацию о том, какое конкретно подразделение компании выполняет то или иное действие при обслуживании заказа.

Итак, диаграммы деятельности представляют собой описание этапов, необходимых для реализации операций. Этот вид диаграмм особенно полезен на ранних этапах проектирования, так как с их помощью разработчики могут изучать алгоритмы и технологические процессы. Однако следует отметить, что диаграммы деятельности представляют вспомогательное средство в рамках объектно-ориентированного подхода и не должны использоваться для разработки программного обеспечения по блок-схемам.

Упражнения

1. Разработайте диаграмму вариантов использования для системы контроля покупок в книжном магазине. Система обеспечивает сканирование каждой книги и вычисляет общую сумму покупки. Если штрих-код на книге поврежден, то кассир вводит стоимость книги вручную. Система позволяет клиенту расплатиться наличными или по кредитной карте. После подтверждения оплаты система контроля покупок печатает чек и сохраняет данные о сделанных покупках. Если клиент хочет вернуть ранее купленный товар в магазин, то предъявляет чек, подтверждающий сделанную покупку. Кассир принимает товары, вводит в систему отмену покупки по предъявленному чеку и выдает наличные клиенту. При этом система ведет учет возвращенных покупок. Все приведенные на диаграмме варианты использования должны содержать описание, включающее основной и альтернативный поток событий.
2. Разработайте диаграмму вариантов использования для компьютерной системы электронной почты. Каждый пользователь системы должен иметь возможность отправлять почту с любого компьютера и получать почту по одной, принадлежащей ему учетной записи. Если пользователь не зарегистрирован в системе или хочет создать новую учетную запись, система должна предоставить ему эту возможность. Так же система должна предоставлять средства для ответа на письма и для их пересылки, а так же для сохранения сообщений в файл и их распечатки. Кроме того, пользователи должны иметь возможность отправлять письма сразу нескольким другим пользователям при помощи списков рассылки. Все приведенные на диаграмме варианты использования должны содержать описание, включающее основной и альтернативный поток событий.

3. Разработайте диаграмму вариантов использования для задачи об интернет-магазине (упражнение 9 из раздела «Моделирование классов»).
4. Для задачи о системе распространения электронной почты в компьютерной сети из предыдущего раздела, разработать диаграмму последовательности создания учетной записи для пользователя электронной почты.
5. Для задачи о системе распространения электронной почты в компьютерной сети из предыдущего раздела, разработать диаграмму последовательности отправки письма по электронной почте.
6. Разработайте диаграмму последовательности для задачи об интернет-магазине (упражнение 9 из раздела «Моделирование классов»).
7. Для описанной ниже задачи разработайте диаграмму деятельности. Распределите ответственность при помощи плавательных дорожек. Покажите взаимодействия между подразделениями. Пользователь принимает решение модернизировать свой компьютер и купить DVD-проигрыватель. Он начинает со звонка в отдел продаж производителя компьютера, откуда его направляют в службу поддержки. Служба поддержки сообщает пользователю о моделях и возможных вариантах установки DVD-проигрывателя. Пользователь выбирает модель проигрывателя и заказывает доставку почтой. Он получает проигрыватель, успешно устанавливает его в компьютер, после чего отправляет по почте оплаченный счет.
8. Подготовьте диаграмму деятельности, описывающую в подробностях процесс входа в систему электронной почты. При разработке диаграммы следует обратить внимание на то, что ввод пароля и имени пользователя может осуществляться в произвольном порядке.
9. Для описанной ниже задачи разработайте диаграмму деятельности. Покажите зоны ответственности при помощи плавательных дорожек. Покажите изменения состояния проекта по мере выполнения деятельности.

   Компания производит новый продукт и должна координировать несколько отделов. Продукт начинает свое существование с маркетинговой идеи, которая передается в проектный отдел. Проектный отдел моделирует функции продукта и подготавливает проект. Отдел производства изучает и корректирует проект, приводя его в соответствие с имеющимся оборудованием. Проектный отдел принимает изменения, после чего проект изучает служба поддержки: хороший проект должен подразумевать удобство ремонта. Проектный отдел принимает предложения службы поддержки и проверяет, что после корректировок проект удовлетворяет требованиям, предъявленным к целевой функциональности.
10. Разработайте диаграмму деятельности для задачи об интернет-магазине (упражнение 9 из раздела «Моделирование классов»).

5. Физическое представление модели

Все рассмотренные выше диаграммы отражали концептуальные аспекты моделирования структуры и поведения системы и относились к логическому уровню представления, основное назначение которого состоит в анализе структурных и функциональных отношений между элементами моделируемой системы.

Особенность логического представления заключается в том, что оно оперирует понятиями, которые не имеют материального воплощения. Для того чтобы можно было рассуждать о желаемом поведении системы, создаются диаграммы прецедентов. Словарь предметной области описывается с помощью диаграмм классов. Чтобы показать, как описанные в словаре сущности совместно работают для обеспечения нужного поведения, используются диаграммы последовательностей, состояний и деятельности. В конечном счете, логические чертежи должны превращаться в реальные вещи, например в исполняемые программы, библиотеки, таблицы, файлы и документы.

Другими словами, различные элементы логического представления, такие как классы, ассоциации, состояния, сообщения не существуют материально или физически. Однако для создания конкретной физической системы необходимо некоторым образом реализовать все элементы логического представления в виде конкретных материальных сущностей.

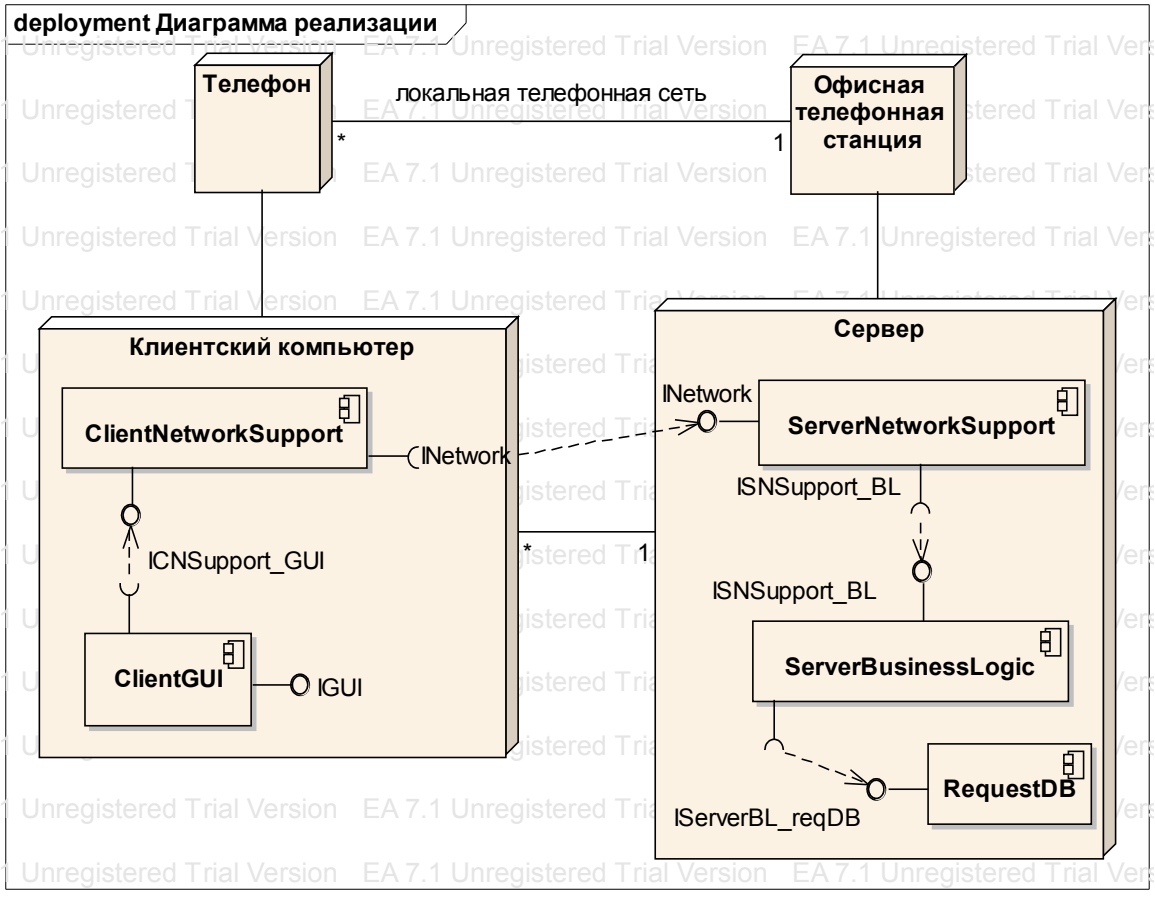


Рис. 5.1. Пример размещения компонент на диаграмме развертывания

Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно – ***физический уровень представления.***

Полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой.

В языке UML для моделирования физического представления проектируемой системы используются так называемые диаграммы реализации (рис. 5.1), которые включают в себя две отдельные канонические диаграммы: ***диаграмму компонентов и диаграмму развертывания.***

5.1. Диаграммы компонентов

Диаграммы компонентов используются для моделирования статического вида системы с точки зрения реализации. Этот вид диаграмм в первую очередь связан с управлением конфигурацией частей системы, составленной из компонентов, которые можно соединять между собой различными способами.

Диаграмма компонентов (component diagram) – описывает особенности физического представления разрабатываемой системы, позволяя определить ее архитектуру, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Данная диаграмма обеспечивает согласованный переход от логического к физическому представлению системы в виде программных компонентов.

Диаграммы компонентов показывают, как выглядит модель на физическом уровне, изображая компоненты программного обеспечения и связи между ними. То есть основными элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними. Кроме этого, на ней могут отображаться ключевые классы, входящие в компоненты. По существу, диаграммы компонентов – это не что иное, как диаграммы классов, сфокусированные на системных компонентах.

5.1.1. Компонент

Компонент (component) – физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а так же функционального поведения моделируемой программной системы.

Можно выделить два основных типа компонентов: компоненты рабочие продукты и исполняемые компоненты.

Компоненты рабочие продукты (work product component) – это компоненты, которые не принимают непосредственного участия в работе исполняемой системы, но при этом являются рабочими продуктами, из которых данная исполняемая система создается. Сюда можно отнести файлы с

исходными текстами программ и данные, из которых создается исполняемая система.

Компоненты исполнения (execution component) – это компоненты, которые необходимы и достаточны для построения исполняемой системы. К их числу относятся динамически подключаемые библиотеки (.dll) и исполняемые программы (.exe).

Для графического представления компонента используется специальный символ - прямоугольник, в правом верхнем углу которого изображена пиктограмма в виде небольшого прямоугольника со вставленными слева двумя маленькими прямоугольниками.

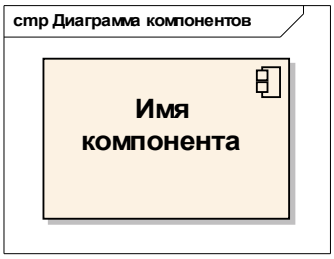


Рис. 5.2. Графическое изображение компонента

Это каноническое обозначение позволяет визуализировать компонент без привязки к конкретной операционной системе или языку программирования.

Внутри прямоугольника записывается имя компонента. Имя компонента может состоять из любого числа букв цифр и некоторых знаков препинания за исключением, таких как двоеточия, которые применяются для отделения имени компонента от имени объемлющего пакета.

В качестве имен компонентов принято использовать имена исполняемых файлов, динамических библиотек, Web-страниц, текстовых файлов или файлов справки, файлов баз данных или файлов с исходными текстами программ и другие.

Для более детальной спецификации компонентов кроме имени можно указать и некоторую дополнительную информацию в виде стереотипа. Компоненты могут иметь следующие стандартные стереотипы:

- ***«file»*** - любой файл кроме таблицы;
- ***«executable»*** – программа (исполняемый файл);
- ***«library»*** – статическая или динамическая библиотека;
- ***«document»*** – остальные файлы (например, файл справки);
- ***«table»*** – таблица базы данных.

5.1.2. Отношения между компонентами

Между отдельными компонентами изображают взаимодействия, соответствующие зависимостям на этапе компиляции или выполнения программы.

Отношение зависимости (dependency) - это отношение, изображаемое на диаграмме компонентов, которое означает зависимость реализации одних компонентов от реализации других.

Данное отношение уже рассматривалось при изучении моделирования классов. Напомним, что изображается зависимость в виде пунктирной линии со стрелкой.

При использовании отношений зависимости на диаграммах компонентов в одних случаях они могут отражать связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода, в других – зависимость может указывать на наличие классов, которые используются в компоненте для создания соответствующих объектов.

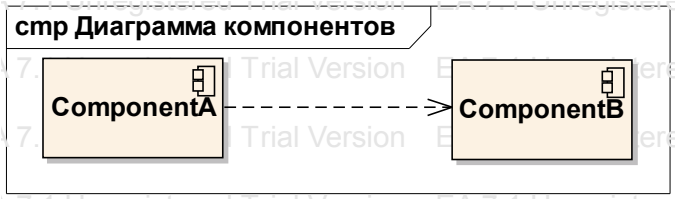


Рис. 5.3. Изображение зависимости между компонентами

Проще говоря, если «ComponentA» зависит от «ComponentB», это значит, что A не может быть скомпилирован до B, то есть сначала компилируется B, а затем уже A. Также любые изменения произведенные в «ComponentB» повлияют на «ComponentA».

Таким образом, с помощью диаграммы компонентов можно оценить последствия любых вносимых изменений и если какой-то компонент зависит от большего числа других компонентов, велика вероятность того, что его затронут изменения в системе.

И, наконец, зависимости дают понять, какие части системы можно использовать повторно, а какие нет. Очевидно, что «ComponentA» будет трудно применить во второй раз. Поскольку он зависит от «ComponentB», то сделать это можно только совместно с «ComponentB». С другой стороны, «ComponentB» легко использовать повторно, так как он ни от чего не зависит. Это значит, чем от меньшего числа компонентов зависит данный компонент, тем легче его будет использовать повторно.

Важно отметить, что в проектируемой модели не должно быть циклов по отношению зависимости. Все циклические зависимости необходимо устранить до начала генерации кода.

В качестве примера, иллюстрирующего применение отношения зависимости между компонентами, на рис. 5.4 показан фрагмент диаграммы компонентов, описывающей пять исходных файлов и зависимости между ними

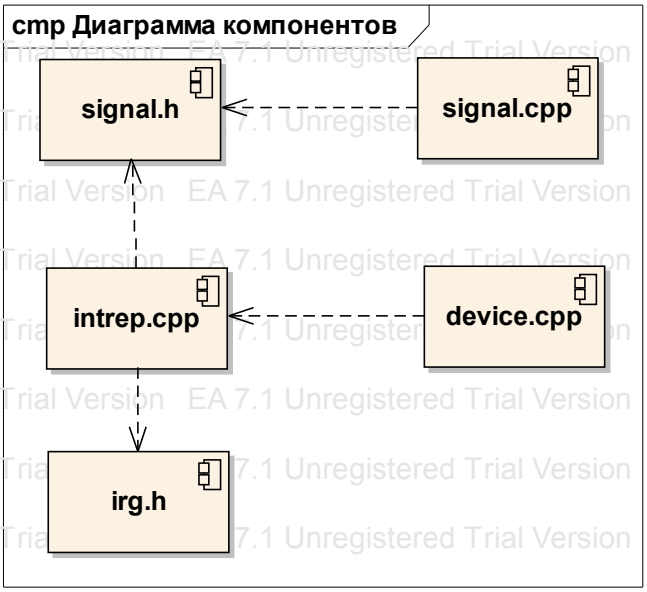


Рис. 5.4. Пример фрагмента диаграммы компонентов

В приведенном примере заголовочный файл *signal.h* используется двумя другими файлами *interp.cpp* и *signal.cpp*. Один из них *interp.cpp* зависит при компиляции от другого заголовочного файла *irq.h*. В свою очередь файл *device.cpp* зависит от *interp.cpp*.

При наличии такой диаграммы компонентов легко проследить что произойдет при изменениях. Так, например, изменение исходного файла *signal.h* потребует перекомпиляции трех других файлов: *signal.cpp, inter.cpp* и *device.cpp*. Из той же диаграммы видно, что файл *irg.h* такое изменение не затронет.

5.1.3. Зависимость между компонентами и классами

Во многих отношениях компоненты подобны классам, так как каждый класс или подсистема модели может быть преобразована в компонент исходного кода, который после создания добавляется к диаграмме компонентов. Однако между ними есть и существенные различия.

Классы представляют собой логические абстракции, а компоненты – физические сущности. Классы могут обладать атрибутами и операциями, а компоненты обладают только операциями, доступными через их интерфейсы.

Отношение между компонентом и классом, который он реализует, может быть изображено явно в виде отношения зависимости (рис. 5.5).

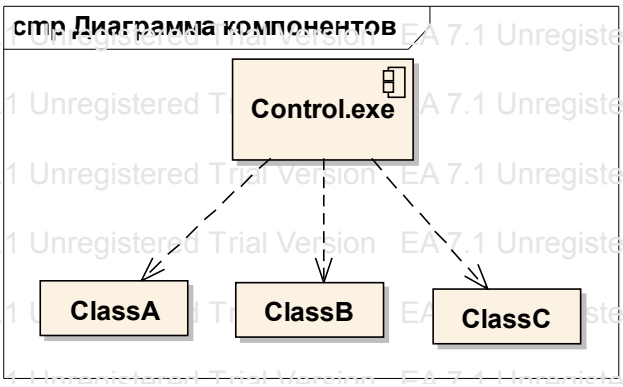


Рис. 5.5. Графическое изображение зависимости между компонентом и классами.

Такая информация имеет значение для обеспечения согласованности логического и физического представлений модели системы. Это значит, что изменения в структуре описаний классов могут привести к изменению данной зависимости. Однако, как правило, подобные отношения не визуализируются, а хранятся как часть спецификации компонента.

Внешнее представление компонента, в котором его внутренняя структура полностью скрыта от его окружения, получило специальное название – «черный ящик». В этом случае открытыми или доступными извне являются только интерфейсы компонента, которые образуют основу для взаимосвязей компонентов между собой.

5.1.4. Интерфейс

Интерфейс (interface) – это внешне видимый, именованный набор операций, который класс, компонент или подсистема может предоставить другому классу, компоненту или подсистеме, для выполнения им своих функций. Он позволяет компонентам скрыть их внутреннее устройство и предоставить окружению определенный способ обращения к своим функциям.

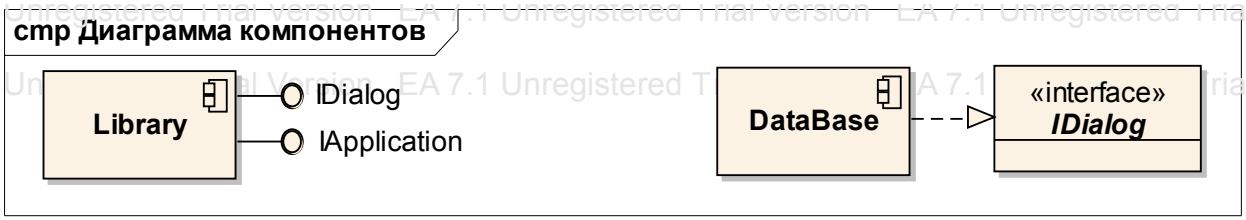


Рис. 5.6. Графическое изображение интерфейсов на диаграмме компонентов.

В общем случае интерфейс изображается окружностью, которая соединяется с компонентом сплошной линией. Интерфейс может также иметь имя, которое записывается рядом с символом интерфейса. При этом имя интерфейса рекомендуется начинать с заглавной буквы « I ».

Семантически данный символ означает реализацию интерфейса, а наличие нескольких символов означает, что данный компонент реализует соответствующий набор интерфейсов.

Кроме того, интерфейс может быть изображен в виде прямоугольника класса со стереотипом «interface» и секцией поддерживаемых операций. В этом случае он связывается с соответствующим компонентом отношением ***реализации (realize)*,** которое изображается в виде пунктирной линии с закрашенной стрелкой, и означает, что данный компонент реализует соответствующий интерфейс. Такой вариант используется, когда необходимо представить внутреннюю структуру данного интерфейса.

Интерфейсы в основном применяются для соединения между собой различных видов компонентов. Для этого компонент, использующий интерфейс, присоединяют к нему с помощью символа зависимости, то есть пунктирной линией со стрелкой. Это значит, что данный компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения.

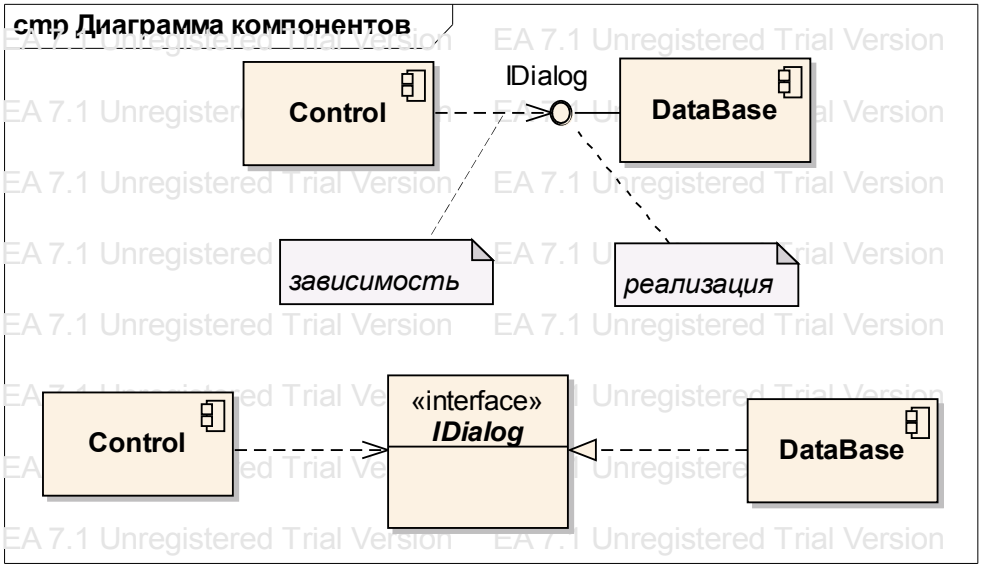


Рис. 5.7. Графическое изображение компонентов и связывающих их интерфейсов.

На рис. 5.7 представлены два способа изображения отношений между компонентами. Первый способ не показывает внутреннее устройство интерфейса, а второй способ представляет интерфейс в развернутом виде и дает возможность показать внутреннюю структуру данного интерфейса, добавив в него соответствующие операции.

Отношения зависимости и реализации, используемые на данной диаграмме компонентов, отображают информацию о том, что компонент *«Control»* зависит или от интерфейса *«IDialog»*, который реализуется компонентом *«DateBase»*.

Таким образом, компонент может иметь две разновидности интерфейсов: предоставляемые и требуемые, которые в нотации языка UML имеют собственные символы визуализации на диаграммах компонентов.

Предоставляемый интерфейс (provided interface) – интерфейс, который компонент предлагает (реализует) для своего окружения. Предоставляемый интерфейс может быть реализован непосредственно компонентом или одним из классов, который находится в собственности этого компонента. Графически предоставляемые интерфейсы изображаются сплошной линией с кружком на конце. Это обозначение получило название «леденец на палочке».

Требуемый интерфейс (required interface) – интерфейс, который необходим компоненту от своего окружения для выполнения заявленной функциональности. Требуемые интерфейсы компонент запрашивает у других компонентов, которые их реализуют. Требуемые интерфейсы в языке UML изображаются в виде сплошной линии с половинкой окружности на конце.

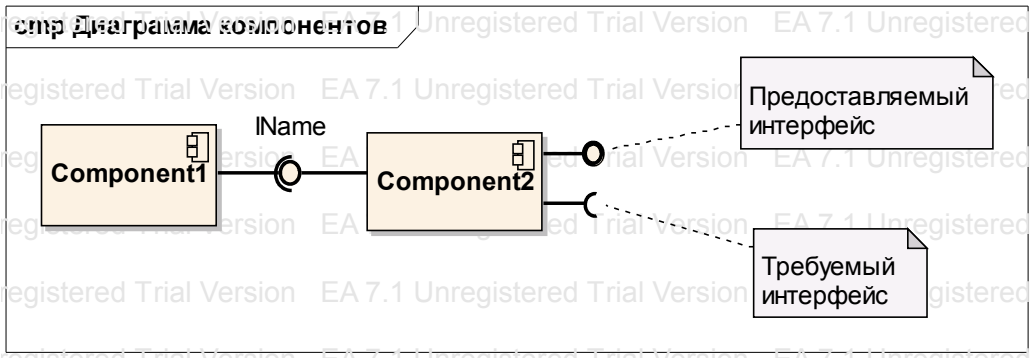


Рис. 5.8. Графическое изображение компонента с интерфейсами

Такое обозначение для предоставляемого и требуемого интерфейса выбрано не случайно, так как оно специфицирует возможность соединить требуемый и предоставляемый интерфейс в некоторую связь, обеспечивающую коммуникацию между двумя и более компонентами.

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программ, не изменяя другие, так как компонент является заменяемой единицей, которая может быть заменена во время проектирования другим компонентом, предлагающим эквивалентную функциональность, основанную на совместимости интерфейсов. Иначе говоря, совместимость предоставляемых и требуемых интерфейсов при их соединении обеспечивает то, что существующий компонент в системе может быть заменен другим, который предлагает такое же множество сервисов. Таким образом, информация, представленная диаграмме компонентов может со временем

меняться, так как могут уточняться интерфейсы, добавляться новые компоненты и заменяться уже существующие.

Одним из основных назначений диаграмм компонентов является моделирование конфигурации файлов, содержащих исходный код, из которых затем создаются исполняемые файлы. Графическое моделирование исходного кода особенно полезно для визуализации зависимостей между различными файлами при компиляции, а так же для управления разбиением файлов на отдельные независимые группы, которые в последствии могут объединяться в ходе ветвления процесса разработки.

Итак, с помощью диаграмм компонентов можно визуализировать исходные файлы и связи между ним. Для этого сначала определяются представляющие интерес наборы исходных файлов и моделируются в виде компонентов со стереотипом «*file*». Затем устанавливаются взаимосвязи между рассматриваемыми компонентами, возникающие при компиляции, которые изображаются с помощью отношений зависимости. По мере увеличения модели многие файлы с исходными текстами можно объединять в группы. В UML для моделирования групп исходных файлов используются пакеты.

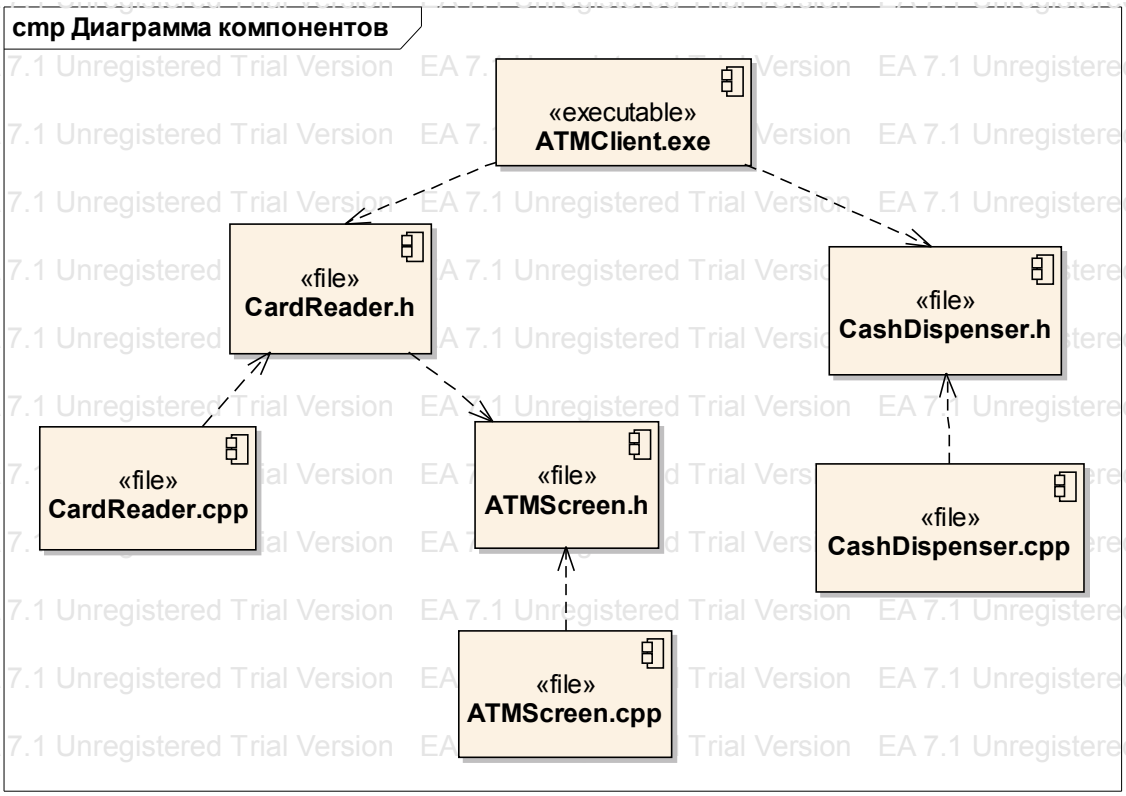


Рис. 5.9. Диаграмма компонентов для клиента ATM.

На рис. 5.9 изображена одна из диаграмм компонентов для системы ATM (банкомат). На этой диаграмме показаны компоненты клиента системы ATM. Система разрабатывается на языке C++. У каждого класса имеется свой

собственный заголовочный файл и файл тела класса. Например, класс ATMScreen преобразуется в два соответствующих компонента ATMScreen.h и ATMScreen.cpp диаграммы компонентов.

Между компонентами установлены определенные зависимости, которые соответствуют зависимостям, возникающим при компиляции. Например, класс CardReader зависит от класса ATMScreen. Это значит, что, для того, чтобы класс CardReader мог быть скомпилирован, класс ATMScreen должен уже существовать.

После компиляции всех файлов создается исполняемый файл ATMClient.exe, который сам по себе является исполняемой программой или артефактом.

У системы может быть несколько диаграмм компонентов, в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема является совокупностью или пакетом компонентов. В данном случае ATMClient представляет собой отдельный пакет компонентов. Другой пакет компонентов может быть разработан для ATMServer.

Разные пакеты могут размещаться по разным узлам проектируемой системы, поэтому диаграмма компонентов, как правило, разрабатывается совместно с диаграммой развертывания, на которой представляется информация о физическом размещении компонентов программной системы по ее отдельным узлам.

5.2. Диаграмма развертывания

Физическое представление программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Если создается простая программа, которая может выполняться локально на компьютере пользователя, не используя никаких распределенных устройств и сетевых ресурсов, то нет необходимости разрабатывать диаграмму развертывания. Однако при создании распределенных или клиент-серверных приложений требуется визуализировать сетевую инфраструктуру программной системы.

Диаграмма развертывания (deployment diagram) предназначена для представления общей конфигурации или топологии распределения программной системы и содержит изображение размещения различных артефактов (исполняемых компонентов и динамических библиотек) по отдельным узам системы.

Диаграмма развертывания визуализирует только те элементы физического представления модели, которые существуют во время выполнения или исполнения программной системы, например, исполняемые компоненты. Те элементы, которые не используются на этапе выполнения на диаграмме развертывания, как правило, не показываются. Так, например, компоненты с исходными текстами программ могут присутствовать на диаграмме

компонентов, а на диаграмме развертывания они не указываются, а может указываться только исполняемый компонент, получаемый в результате их компиляции.

Следует отметить, что в отличие от диаграмм логического представления и диаграмм компонентов, диаграмма развертывания проектируется, как правило, в единственном экземпляре, так как она должна всецело отражать особенности топологии и реализации разрабатываемой системы.

Итак, на диаграммах развертывания визуализируется конфигурация обрабатывающих узлов, на которых выполняется программная система и компонентов, размещенных в этих узлах, а так же специфицируются взаимосвязи между отдельными узлами в виде отношений ассоциации и зависимости.

5.2.1. Узел

Узел (node) – представляет собой физически существующий элемент системы, который может обладать вычислительным ресурсом или являться техническим устройством.

На диаграммах развертывания узлы обычно служат для представления аппаратных устройств или исполняемых программных средств. С каждым узлом ассоциируется некоторое множество элементов, которые на нем развертываются. Развернутые на узлах элементы представляют собой артефакты.

В графической нотации языка UML узел изображается в форме куба, внутри которого указывается имя узла. Сами узлы могут представляться или в качестве типа или в качестве экземпляра.

В первом случае имя узла записывается с заглавной буквы в форме *<Имя типа узла>* и не подчеркивается. Такое имя указывает на некоторую разновидность узлов, присутствующих в модели системы.

Во втором случае имя экземпляра записывается в форме *<имя узла : Имя типа узла>*. При этом собственное имя узла записывается с маленькой буквы и вся запись подчеркивается.

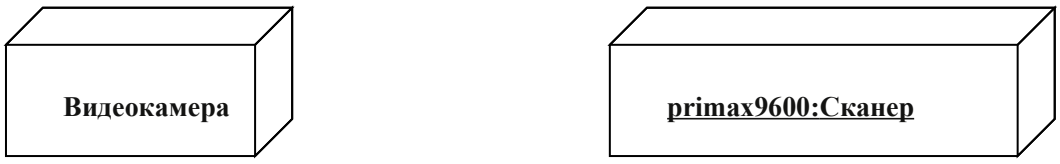


Рис. 5.10. Графическое изображение узла на диаграмме развертывания

На представленном рисунке узел с именем *Видеокамера* относится к общему типу и никак не конкретизируется. Второй узел является узлом-экземпляром конкретной модели сканера.

В качестве дополнения к имени узла могут использоваться различные стереотипы, которые явно специфицируют назначение данного узла. Для этой цели применяются следующие текстовые стереотипы: «server» (сервер),

«workstation» (рабочая станция), «device» (устройство), «sensor» (датчик), «modem» (модем), «net» (сеть) и другие, смысл которых понятен из контекста. При этом в некоторых CASE-системах данные стереотипы изображаются не в текстовой форме, а в виде рисунка, например, рабочую станцию можно изобразить в виде компьютера, а консоль в виде рисунка клавиатуры.

5.2.2. Путь коммуникации

Между отдельными узлами ни диаграмме развертывания можно указать взаимодействия или пути коммуникации.

Путь коммуникации (communication path) является ассоциацией между двумя узлами, посредством которой они могут обмениваться сигналами и сообщениями.

Графически путь коммуникации изображается в виде обычной ассоциации. При этом путь коммуникации может иметь дополнительные спецификации, которые определяются согласно общему определению ассоциации.

На рис. 5.11. изображена диаграмма развертывания для телефонной службы приема заявок. Данная диаграмма показывает, из каких типов узлов будет состоять телефонная служба приема заявок, а так же какие пути коммуникации с пометками множественности (кратности) определены между ними.

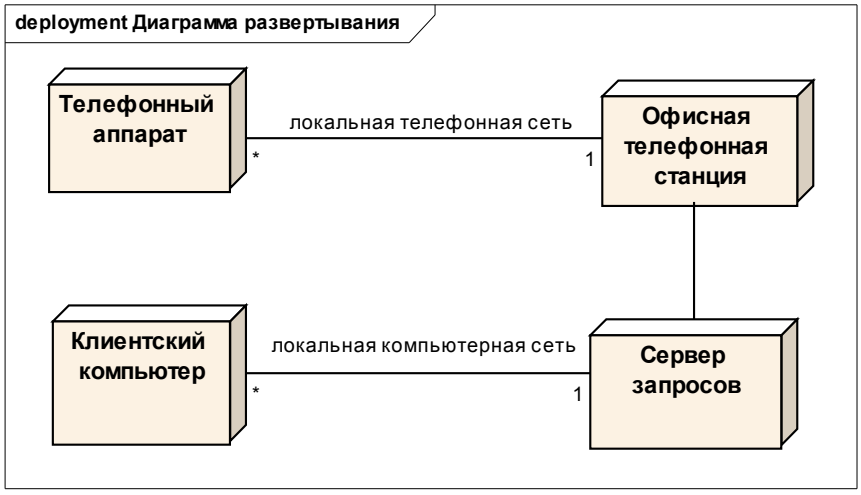


Рис. 5.11. Пути коммуникации между различными узлами.

Как уже говорилось выше, диаграммы развертывания содержат не только физические узлы проектируемой системы, но и могут отображать размещение различных компонентов по отдельным узлам системы или ***развертывание***.

В общем случае развертывание представляет собой размещение одного или более компонентов на отдельном узле с необязательной спецификацией.

Размещение компонентов на узлах диаграммы развертывания может быть изображено следующими способами.

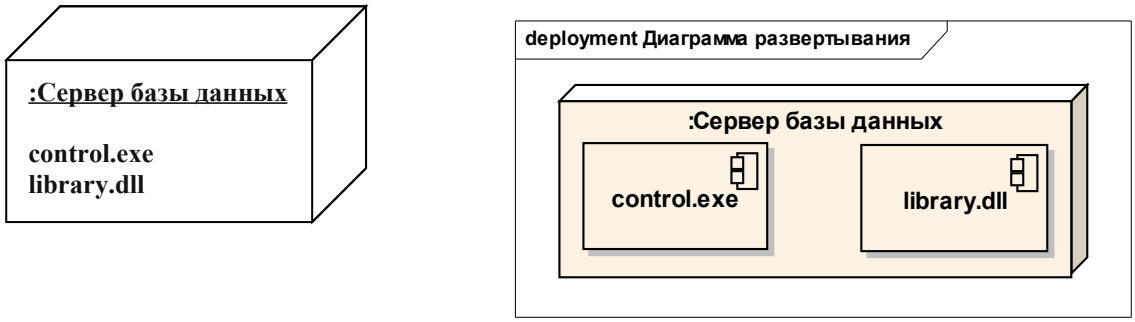


Рис. 5.12. Способы представления развертывания компонентов в узле.

Первый способ позволяет изобразить развертывание компонентов в форме списка артефактов внутри символа узла. Этот способ является самым компактным и потому он наиболее удобен при построении моделей развертывания для больших систем. Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов.

Напомним, что в качестве таких вложенных артефактов могут выступать только исполняемые компоненты и динамические библиотеки.

Следует отметить, все исполняемые компоненты должны быть размещены по соответствующим узлам проектируемой системы, то есть в модели не должно оказаться неразмещенных исполняемых компонентов.

На рисунке 5.13. показана диаграмма развертывания для примера с системой ATM, диаграмма компонентов которой была построена ранее.

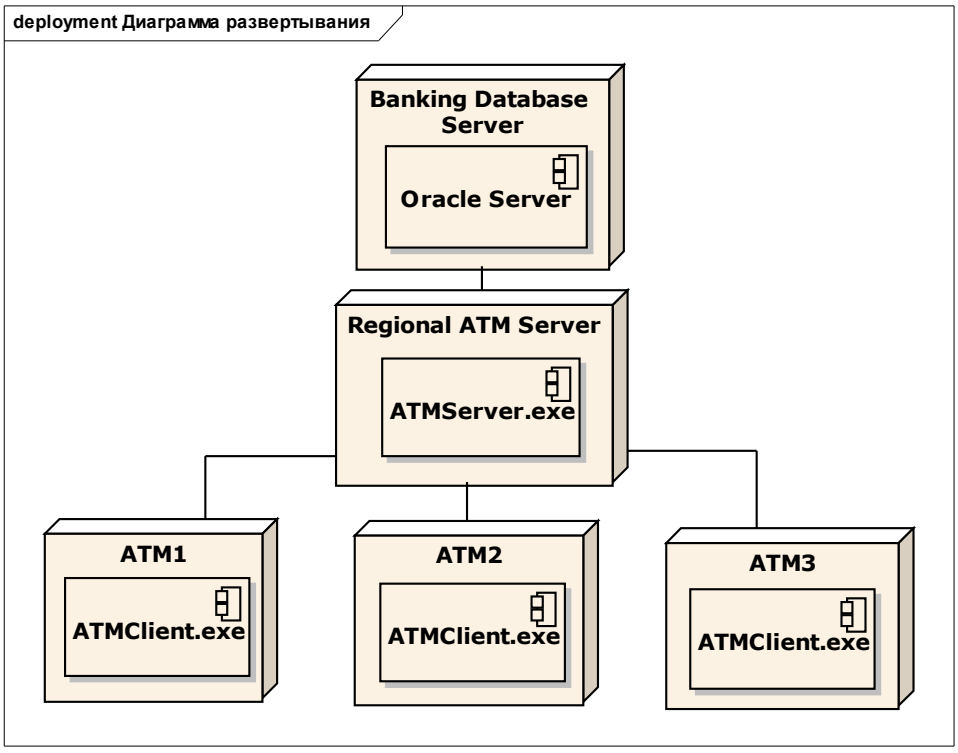


Рис. 5.13. Пример диаграммы развертывания.

Из представленной диаграммы можно узнать о физическом размещении системы ATM. Клиентские программы ATM будут работать в нескольких местах на соответствующих банкоматах. Они будут сообщаться с региональным сервером ATM, на котором будет работать программное обеспечение сервера. В свою очередь региональный север будет сообщаться с сервером банковской базы данных, работающим под управлением Oracle.

Итак, диаграмма развертывания – это один из двух видов диаграмм используемых при моделировании физического представления проектируемой системы. Можно сказать, что диаграммы развертывания располагаются на стыке программных средств и аппаратуры и позволяют судить о топологии процессов и устройств, на которых выполняется разрабатываемая система.

Диаграмма развертывания завершает процесс объектно-ориентированного анализа и проектирования для конкретной программной системы. Разработка этой диаграммы – это, как правило, последний этап спецификации модели.

Литература

[1] Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Дж.Рамбо, А. Якобсон – М.; ДМК, 2000.

[2] Леоненков, А.В. Самоучитель UML. / А.В. Леоненков – СПб.: БХВ-Петербург, 2007.

[3] Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка. / Дж. Рамбо, М. Блаха – СПб.: Питер, 2007.

[4] Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на языке С++. / Г.Буч – М.: Бином, 1999.

[5] Якобсон, А. Унифицированный процесс разработки программного обеспечения. / А.Якобсон, Г. Буч, Дж. Рамбо – СПб.: Питер, 2002.

[6] Вендров, А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. / А.М. Вендров – М.: Финансы и статистика, 2000.

[7] Лафоре, Р. Объектно-ориентированное программирование в С++. / Р. Лафоре. – СПб.: Питер, 2004.

[8] Лагутина, Н. С. Основы объектно-ориентированного программирования на языке С++. / Н.С. Лагутина. – Ярославль: ЯрГУ, 2008.

Содержание