## Тема 4.3. Введение в UML

**4.3.1.** [**Типы диаграмм UML**](http://www.intuit.ru/department/se/vismodtp/3/#sect1)

**4.3.1.** [**Система "Телефонная служба приема заййййййййййййййййййййявок"**](http://www.intuit.ru/department/se/vismodtp/3/#sect2)

**4.3.1.** [**Диаграммы случаев использования (use case diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/#sect3)

**4.3.1.** [**Диаграммы активностей (activity diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/2.html#sect4)

**4.3.1.** [**Диаграммы развертывания (deployment diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/2.html#sect5)

**4.3.1.** [**Диаграммы компонент (component diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/2.html#sect6)

**4.3.1.** [**Диаграммы коммуникаций (communication diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/3.html#sect7)

**4.3.1.** [**Диаграммы последовательностей (sequence diagrams).**](http://www.intuit.ru/department/se/vismodtp/3/3.html#sect8)

**4.3.1.** [**Временные диаграммы (timing diagrams)**](http://www.intuit.ru/department/se/vismodtp/3/3.html#sect9)

**4.3.1. [Диаграммы схем взаимодействия (interaction overview diagram)](http://www.intuit.ru/department/se/vismodtp/3/3.html" \l "sect10)**

**4.3.1.** [**Диаграммы классов (class diagrams)**](http://www.intuit.ru/department/se/vismodtp/4/#sect1)

**4.3.1.** [**Ассоциации (association)**](http://www.intuit.ru/department/se/vismodtp/4/#sect2)

**4.3.1.** [**N-арные ассоциации и класс-ассоциации**](http://www.intuit.ru/department/se/vismodtp/4/2.html#sect3)

**4.3.1.** [**Агрегирование**](http://www.intuit.ru/department/se/vismodtp/4/2.html#sect4)

**4.3.1.** [**Диаграммы пакетов (package diagrams)**](http://www.intuit.ru/department/se/vismodtp/4/3.html#sect5)

**4.3.1.** [**Диаграммы объектов (object diagrams)**](http://www.intuit.ru/department/se/vismodtp/4/4.html#sect6)

**4.3.1.** [**Кооперации (collaborations)**](http://www.intuit.ru/department/se/vismodtp/4/4.html#sect7)

**4.3.1.** [**Диаграммы конечных автоматов (statechart diagrams)**](http://www.intuit.ru/department/se/vismodtp/4/5.html#sect8)

**4.3.1.** [**Ремарки об изучении и использовании UML.**](http://www.intuit.ru/department/se/vismodtp/4/5.html#sect9)

**4.3.1.** [**Краткий обзор литературы об UML**](http://www.intuit.ru/department/se/vismodtp/4/5.html#sect10)

**4.3.1.** [**Контрольные вопросы**](http://www.intuit.ru/department/se/vismodtp/4/5.html#sect11) **по теме «Введение в UML»**

**4.3.1.** [**Тестовые**](http://www.intuit.ru/department/se/vismodtp/3/3.html#sect11) **задания по теме «Введение в UML»**

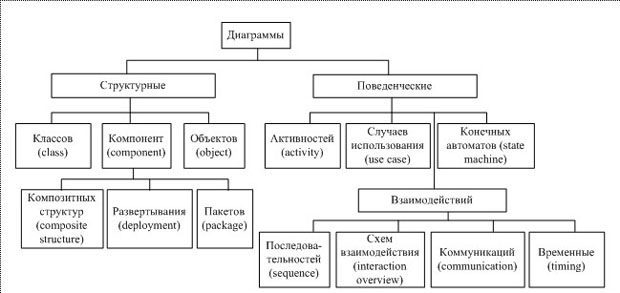
### 4.3.1.Типы диаграмм UML

Познакомимся с одним из визуальных языков – UML (Unified Modeling Language), который является на данный момент самым распространенным и общеизвестным способом моделирования программного обеспечения.

«Скелетом» UML является диаграммная структура UML, в котором можно выделить следующие типы диаграмм (diagram types) ([рис.4 3-1](http://www.intuit.ru/department/se/vismodtp/3/#image.3.1)):

* Структурные диаграммы:
* диаграммы классов (class diagrams) предназначены для моделирования структуры объектно-ориентированных приложений - классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;
* диаграммы компонент (component diagrams) используются при моделировании компонентной структуры распределенных приложений; внутри каждая компонента может быть реализована с помощью множества классов;
* диаграммы объектов (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;
* диаграммы композитных структур (composite structure diagrams) используются для моделирования составных структурных элементов моделей - коопераций, композитных компонент и т.д.;
* диаграммы развертывания (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);
* диаграммы пакетов (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.
* Поведенческие диаграммы:
  + диаграммы активностей (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;
  + диаграммы случаев использования(use case diagrams) предназначены для "вытягивания" требований из пользователей, заказчика и экспертов предметной области;
  + диаграммы конечных автоматов (state machine diagrams) применяются для задания поведения реактивных систем;
  + диаграммы взаимодействий (interaction diagrams):
* диаграммы последовательностей (sequence diagrams) используются для моделирования временных аспектов внутренних и внешних протоколов ПО;
* диаграммы схем взаимодействия (interaction overview diagrams) служат для организации иерархии диаграмм последовательностей;
* диаграммы коммуникаций (communication diagrams) являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);
* временные диаграммы (timing diagrams) являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

На [рис. 4.3-1](http://www.intuit.ru/department/se/vismodtp/3/#image.3.1) не все узлы обозначают типы диаграмм - некоторые изображают лишь группы диаграмм, например, "Структурные", "Поведенческие", "Взаимодействий".



**Рис. 4.3-1.**  Типы диаграмм UML 2.0

Описание нотации UML структурировано по разным типам диаграмм, хотя они и не являются строго обязательными. Различные конструкции языка можно вставлять в разнотипные диаграммы. Например, экземпляры классов можно изображать на одной диаграмме с самими классами, и пакеты также могут показываться на диаграммах классов. Таким образом, границы между различными типами диаграмм размываются. Создание диаграмм того или другого типа - всего лишь наиболее устоявшийся, традиционный способ использования UML, не исключающий, однако, и других вариантов.

***Система "Телефонная служба приема заявок"***

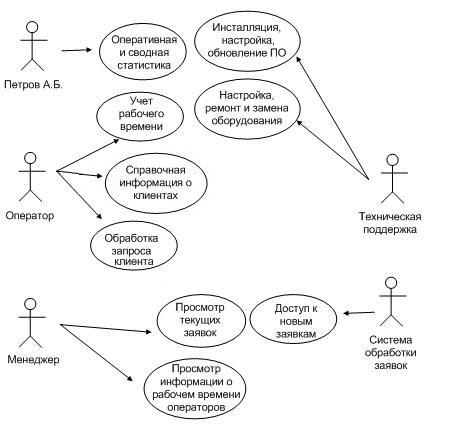
В дальнейшем в качестве примера будет использоваться система "Телефонная служба приема заявок". Далее будут приведены фрагменты этой системы, изображенные с помощью различных UML-диаграмм. Эти примеры будут снабжены некоторыми "сюжетами" - гипотетическими ситуациями процесса разработки, в которых могла появиться необходимость в создании этих диаграмм. Разумеется, приводимые сюжеты далеко не единственные, даже в рамках разработки данной системы. Но они нужны, чтобы с первых же шагов при знакомстве с UML не появлялось чувство пустоты, подвешенных в воздухе иллюстраций.

Часть примеров будет на русском языке, а часть - на английском (это касается всех дальнейших примеров, а не только тех, которые относятся к телефонной службе приема заявок). Если диаграммы связаны с программным кодом, то есть моделируют какие-либо его абстракции (классы, таблицы баз данных, компоненты и пр.), то используется англоязычная терминология - названия модельных сущностей должны быть идентификаторами в программном коде. Если же такого нет, то при именовании используется обычный русский язык.

Итак, заказчик данной системы - это компания, владеющая сетью продуктовых магазинов. Данная компания, кроме обычной розничной торговли, хочет предоставлять еще и сервис по обслуживанию клиентов по телефонным заявкам. Клиент регистрируется в компании, а потом по телефону, в удобное для себя время, делает заказ товаров, которые к нему привозят домой, и он расплачивается. Для этого компания хочет организовать у себя локальный телефонный центр, состоящий из офисной многоканальной АТС, штата операторов и соответствующего программного обеспечения. При этом в компании уже есть информационная система по обработке заявок от постоянных мелкооптовых клиентов, и заказываемая система должна быть с ней проинтегрирована.

***Диаграммы случаев использования (use case diagrams)***

Первым шагом по реализации описанной выше задачи является уточнение требований. Для этого можно применить диаграммы случаев использования UML. Пример такой диаграммы представлен на [рис. 4.3-2.](http://www.intuit.ru/department/se/vismodtp/3/#image.3.2)



**Рис. 4.3-2.**  Пример диаграммы случаев использования

На ней обозначено следующие виды пользователей - оператор, менеджер и представители технической поддержки. Система должна также поддерживать внешний интерфейс с системой обработки заявок. Это - четвертый пользователь. Еще одним пользователем системы является Петров А.Б. - директор департамента сбыта товаров, который хочет периодически отслеживать деятельность телефонной службы приема заявок. Для него создано специальное пользовательское место с экранными формами статистики. Случаи использования с [рис.4.3.2.](http://www.intuit.ru/department/se/vismodtp/3/#image.3.2) комментировать не будем, считая, что и так все понятно из картинки.

Различные пользователи ПО, изображаемые на диаграммах случаев использования, назваются актерами (actors). Актеры могут обозначать:

* типовых пользователей ("Менеджер", "Оператор", "Техническая поддержка") - работников компании, сгруппированных по исполняемым обязанностям;
* другие системы, взаимодействующие с данной ("Система обработки заявок");
* выделенного пользователя ("Петров А.Б.").

Отметим, что выделенный пользователь существенно отличается от типового пользователя. Он, как правило, Важная Персона, и согласование функциональности для него согласуется лично с ним. Часто он влияет на оплату проекта, от его мнения о системе, во многом, зависит ее успешная сдача. Такие персоны, ради успеха проекта, нужно уметь идентифицировать и в рамках всей системы создавать некоторую функциональность специально для них и очень при этом стараться!

Случай использования (use case) - это независимая часть функциональности системы, обладающая результирующей ценностью для ее пользователей.

"Независимость" означает, что если случай использования всегда исполняется вместе с некоторым другим, то, по всей видимости, один из них нужно включить в другой (какой именно в какой, как назвать получившийся в итоге случай использования - зависит от обстоятельств).

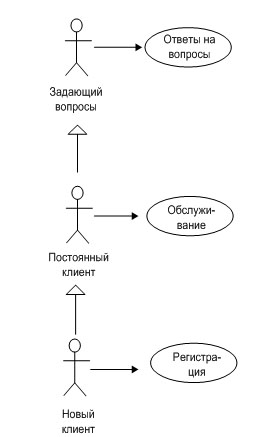
"Результирующая ценность" случая использования для актера системы подразумевает, что он, данный случай использования, должен приносить актеру некоторый законченный и ценный с точки зрения его бизнеса результат. Будучи реализован системой, этот случай использования действительно делает бизнес актера эффективнее, производительнее. Тем самым разработка системы фокусируется на бизнес-целях, а незначительные случаи использования игнорируются. Строится не абстрактная модель функций системы, а набор самых важных (для заказчика и пользователей) сервисов, чтобы каждый из них правильно понять и не один не упустить. И в дальнейшем контроль разработки системы будет осуществляться именно в терминах этого самого важного - того, что нужно заказчику и пользователям.

Казалось бы, что может быть проще - реализовать набор функций, необходимых пользователю. Однако на деле программный проект может незаметно потерять эту цель. Вместо этого можно, например, очень долго заниматься разработкой сложной и многофункциональной архитектуры, после реализации которой разработчики обещают, что все пользовательские функции получатся почти сразу же и очень легко. Однако, как правило, оказывается, что это "сразу же" было сильным преувеличением и проект весьма выбивается из расписания, а многие заказанные пользователем функции в этом окружении сделать тяжело или невозможно. Бывает, что чрезмерная ориентация на "внутреннее совершенство" ПО оканчивается для проекта либо крупными неприятностями, либо полным крахом. Однако бывают и другие случаи, когда только такая ориентация впоследствии и спасает проект. Последнее случается, когда система долго развивается и сопровождается, или когда требования к ней внезапно и сильно меняются, или когда на ее основе делаются другие системы. Необходим баланс между внутренним совершенством программного обеспечения и функциональностью, нужной для заказчика и доставленной ему в срок. Разработка в терминах случаев использования - хороший способ контролировать, что процесс создания системы двигается в нужном направлении.

Итак, основной задачей диаграмм случаев использования является получение требований к системе от заказчика и пользователей. Трудность формализации требований связана с тем, что пользователи и заказчики, с одной стороны, а программисты - с другой, являются специалистами в совершенно разных областях. Первым очень не просто понять логику программной разработки и отделить существенное от несущественного, изъясняться ясно и точно. Вторым трудно разобраться в новой для них предметной области и адекватно отразить это свое понимание в программной системе. К тому же программные системы очень часто являются уникальными. Поэтому набор пожеланий заказчика и пользователей нуждается в дополнительной обработке, освобождению от противоречий, коррекции и, наконец, интеграции, дабы стало возможным "покрыть" его некоторой программной системой. Привлекательность и эффективность диаграмм случаев использования заключается в том, что они просты для понимания непрограммистами и в то же время достаточно формальны.

Отметим, что сами по себе случаи использования не гарантируют того, что программисты и заказчик адекватно понимают друг друга - они могут по-разному трактовать эти случаи использования. Однако в первом приближении масштаб и границы системы очерчены. Для того чтобы детализировать случаи использования, может применяться обычный текст (по одному абзацу на каждый случай использования) и/или другие диаграммы UML.

Существует два вида принципиально разных диаграмм случаев использования - для ПО и для всей системы в целом. Ведь, как правило, ПО является частью более крупной системы. Последняя может включать другое ПО, а также некоторый бизнес-процесс. Пользователями такой системы будут различные клиенты системы (бизнес-актеры), поскольку система создается именно для них. А сама система будет предоставлять для них бизнес-случаи использования. Пример диаграммы бизнес-случаев использования для системы обработки телефонных заявок показан на [рис. 4.3-3](http://www.intuit.ru/department/se/vismodtp/3/#image.3.3).



**Рис. 4.3-3.**  Пример диаграммы бизнес-случаев использования

На этом рисунке можно увидеть трех различных клиентов этой системы - постоянного клиента, нового клиента и задающего вопросы (случайного человека, интересующегося услугами магазина, наличием того или иного товара). В общем, для каждого типа клиентов система должна предоставлять разный сервис: для первого типа клиента - возможность сделать заказ (с внесением в базу данных имени клиента, товара, который он заказал, его цены и сроков доставки), для второго - возможность зарегистрироваться (оператор спрашивает у него фамилию, имя, отчество, адрес и пр., персональную информацию и вносит ее в компьютер), для третьего - возможность отвечать на разные вопросы (возможно, со специальными справочниками товаров и пр.). Причем эти актеры наследуют один от другого именно в том порядке, который указан на диаграмме. При наследовании актеров потомок "получает в наследство" все случаи использования своих предков. Таким образом, каждый из этих клиентов может задавать вопросы, а новый клиент, после того как зарегистрировался, может сделать заказ.

Этих бизнес-клиентов можно было бы изобразить и на [рис. 4.3-2](http://www.intuit.ru/department/se/vismodtp/3/#image.3.2), соединив стрелками с оператором (ведь именно через него они взаимодействуют с системой). Но такая диаграмма может вызвать недоумение, хотя некоторые аналитики склонны так делать. Я считаю, что бизнес-актеров лучше изображать на отдельной диаграмме, а на обычной диаграмме случаев использования показывать только пользователей ПО.

***Диаграммы коммуникаций (communication diagrams)***

Продолжим разговор об отдельных деталях работы будущей системы, начатый в предыдущем разделе. В разные моменты разработки (не только при проектировании) может понадобиться прояснение определенных деталей работы системы, в особенности, деталей, находящихся на стыках различных компонент, разрабатываемых различными членами проектной команды или рабочими группами. Побудительные причины для выяснения этих деталей могут быть различными. Например, разработчики одной из компонент вдруг обнаруживают, что они не понимают того контекста, в котором будет работать их компонента. Или тестеровщики находят ошибки, относительно которых автор каждой из компонент, задействованных в этом стыке, утверждает, что она работает правильно. Во всех этих ситуациях целесообразно собрать совещание с присутствием всех заинтересованных сторон. При этом самый заинтересованный - тестеровщик, менеджер, автор компоненты, у которого возник вопрос и т. д. - готовит гипотезу того, как все должно происходить. И эту гипотезу имеет смысл нарисовать в виде UML-диаграммы. В данном случае целесообразно использовать диаграммы коммуникаций.

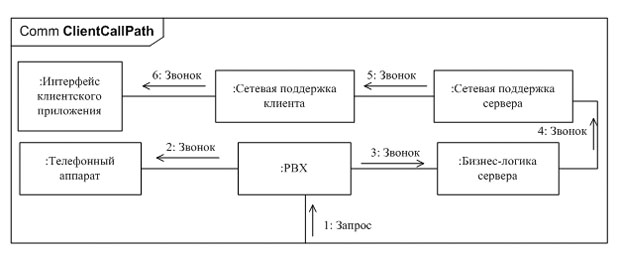
На [рис. 4.3-4](http://www.intuit.ru/department/se/vismodtp/3/3.html#image.3.8) изображается, как выглядит ситуация поступления в систему звонка от клиента. Эта диаграмма может быть полезной в случае, если нужно определить, как информация о звонке распространяется через компоненты ПО, какие процессы при этом происходят в его различных частях и какие данные передаются.

Звонок от клиента приходит на офисную АТС, оттуда уходит на телефонный аппарат свободного оператора и на сервер. От сервера через локальную сеть этот звонок приходит на клиентское ПО того же оператора.

Из этой диаграммы становится понятно, что PBX должен передавать серверу вместе с информацией о звонке еще также информацию и об операторе, с которым он прокоммутировал этого клиента. Ведь сервер должен послать информацию о новом звонке на клиентское ПО именно этого оператора. Получая информацию о звонке, клиентское ПО автоматически открывает оператору специальный диалог, в который тот вводит информацию о звонке прямо во время разговора с клиентом. Еще один важный момент, который следует из этой диаграммы: телефонный звонок на аппарате оператора должен прозвенеть одновременно (или почти одновременно) с появлением на его мониторе диалогового окна для внесения информации о звонке. Все эти вопросы удобно обсуждать на фоне этой диаграммы, хоть на ней и нет всей нужной информации.

На диаграммах коммуникаций изображается взаимодействие ролей классов, компонент, а не конкретные экземпляры. Роли будут подробно обсуждаться в лекции о моделировании систем реального времени. Однако отметим здесь, что роль - более общее понятие, чем объект (экземпляр), и является гнездом, куда могут быть вставлены различные объекты. В синтаксисе UML имена ролей обозначаются без подчеркивания, а имена экземпляров - с подчеркиванием.

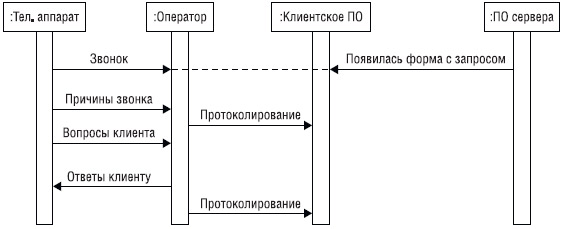
Диаграммы коммуникаций могут использоваться для пояснения кооперации, композитной компоненты или другого композитного объекта (про различные композитные объекты см. следующую лекцию). Поэтому в ней и используются роли, а не объекты. Имя этого композитного объекта указывается в заголовке диаграммы. Там же, в заголовке, используется тег comm для обозначения диаграммы коммуникаций.



**Рис. 4.3-4.**  Пример диаграмм коммуникаций

***Диаграммы последовательностей (sequence diagrams).***

Обратимся теперь к временным свойствам алгоритмов работы системы приема телефонных заявок. Для этого в UML есть диаграммы последовательностей (и еще временные диаграммы, рассматриваемые ниже). Пример такой диаграммы представлен на [рис. 4.3-5](http://www.intuit.ru/department/se/vismodtp/3/3.html#image.3.9).



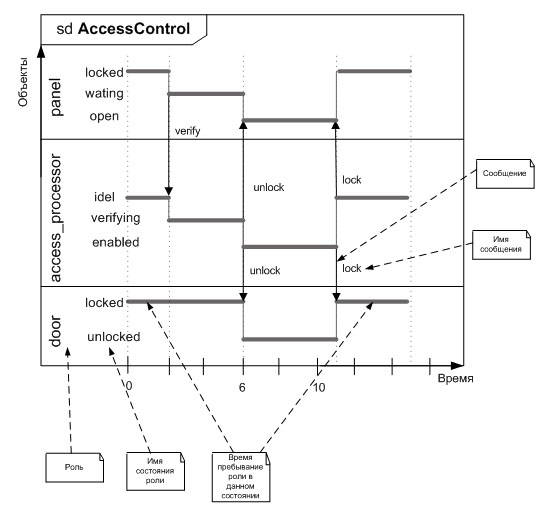
**Рис. 4.3-5.**  Пример диаграммы последовательностей

Данная диаграмма сфокусирована на действиях оператора клиентского ПО. Во-первых, на ней явно изображено, что два события - звонок оператору по телефону и появление диалога для внесения информации о звонке на дисплее оператора - должны происходить одновременно. Это "одновременно" может впоследствии доставить много хлопот, поскольку необходимо будет тестировать это требование в условиях, идентичных условиям заказчика, - в его локальной сети, с тем быстродействием, которое она может обеспечивать, с определенным количеством одновременно работающих в сети операторов и т. д. И понятно, что в этой ситуации ПО должно соревноваться по скорости с процессом коммутации в PBX. Вполне возможно, что телефонный аппарат будет звонить существенно раньше, чем соответствующая экранная форма появится на экране оператора, и это может оказаться весьма неудобным. Значит, нужно "убыстрять" обработку звонка сервером ПО. При этом то или иное быстродействие может потребовать существенно разной реализации серверных компонент, поэтому разумно озаботиться этой проблемой заранее. Создание диаграмм последовательностей помогает на этапе проектирования заметить и не забыть о подобных местах в алгоритмах. Программистам рекомендуется преодолеть нетерпеливость и потратить время на прорисовывание различных деталей архитектуры перед началом программирования, а также во время оного, приступая к новому этапу работы. Вроде бы и так все понятно, но предварительное обдумывание с фиксацией решений с помощью диаграмм, обсуждение этих диаграмм с коллегами может предотвратить ошибки, которые, будучи допущенными, потребуют существенных больших усилий на исправление, много превышающих те, что были потрачены на проектирование.

На диаграммах последовательностей, так же как и на диаграммах коммуникаций, показываются роли классов. Фактически, на обоих диаграммах представлена одна и та же информация, но в разных видах. На диаграммах последовательностей она показана с точки зрения временного аспекта, на диаграммах коммуникаций – с точки зрения отношений взаимодействующих частей (то есть здесь яснее выражен структурный аспект). Можно сказать, что диаграмма последовательностей является двойником диаграмм коммуникаций.

***Временные диаграммы (timing diagrams)***

Этот тип диаграмм является разновидностью диаграмм последовательностей и предназначен для наглядного изображения потока изменения состояний нескольких ролей (классов, компонент[1)](http://www.intuit.ru/department/se/vismodtp/3/footnote.3.1.htm)). Последние изображаются не вертикально, а горизонтально, и основной упор делается на наглядное изображение их состояний, точнее, того, как они меняются во времени. Такая возможность полезна, например, при моделировании встроенных систем.



**Рис. 4.3-6.**  Пример временной диаграммы

На [рис. 4.3-6](http://www.intuit.ru/department/se/vismodtp/3/3.html#image.3.10) показан фрагмент работы системы AccessControl, которая управляет открытием/блокированием двери в помещение по предъявлению человеком электронного ключа. На рисунке показано три компоненты этой системы. Первая, panel, является устройством, у которого есть дисплей для отображения текущего состояния всей системы и устройство считывания электронного ключа. Исходно panel находится в состоянии locked (соответствующая надпись отображается и на дисплее). После того как человек приложил к этому устройству электронный ключ и устройство считало с него информацию, panel посылает эту информацию в виде сообщения verify второй компоненте - процессору (access\_processor) - и переходит в состояние waiting. Процессор до получения сообщения verify находится в состоянии idle, а после получения этого сообщения он переходит в состояние verifying. После успешного окончания проверки данных электронного ключа процессор посылает компонентам panel и door сообщения unlock и переходит в состояние enable. Компонента panel переходит в состояние open. Третья компонента, door (собственно, сама дверь), до этого находилась в состоянии locked и, получив сообщение unlock, открывается (переходит в состояние unlock). Открытой она остается ровно 5 секунд, после чего процессор присылает ей команду lock и она закрывается - снова переходит в состояние locked. Одновременно процессор посылает команду lock также и компоненте panel, которая переходит в свое исходное состояние locked и отображает слово "locked" на дисплее.

Видно, что на временных диаграммах, так же как на диаграммах последовательностей и коммуникаций, показываются только главные ветки алгоритмов, а ветвления отсутствуют. Компоненты и их состояния откладываются по оси ординат, время - по оси абсцисс. Время градуировано в какой-либо шкале измерений. В данном примере каждое деление соответствует двум секундам.

Диаграммная область каждой компоненты - это прямоугольник, отделенный от другого, соседнего (представляющего другую компоненту), прямой линией, параллельной оси абсцисс. Компоненты могут обмениваться сообщениями, с помощью которых происходит синхронизация их поведения. Сообщения изображаются вертикальными линиями со стрелками (вверх или вниз).

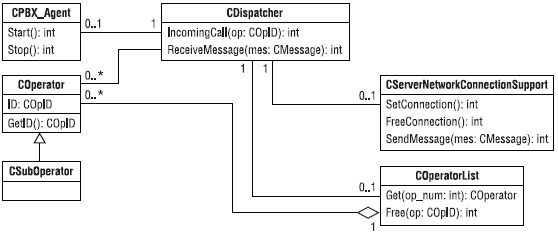
***Диаграммы схем взаимодействия (interaction overview diagram)***

Этот тип диаграмм является смесью диаграмм активностей и диаграмм последовательностей. Вместо действий в узлы диаграмм активностей подставляются диаграммы последовательностей (сценарии). Таким образом, достигается цель задавать сложное поведение с ветвлениями, так как иначе на диаграммах последовательностей ветвления задавать неудобно.

***Диаграммы классов (class diagrams)***

Этот тип диаграмм является основным при разработке объектно-ориентированной системы, так как позволяет наглядно изобразить структуру классов приложения. Такие диаграммы полезны как при предварительном проектировании, так и при рефакторинге, сопровождении и исправлении ошибок, а также при изучении ПО. По этим диаграммам легко генерировать программный код, их легко восстанавливать по уже существующему программному коду. Кроме того, диаграммы классов используются при описании самих языков визуального моделирования, как это будет показано в следующих лекциях.

На [рис. 4.3-7](http://www.intuit.ru/department/se/vismodtp/4/#image.4.1) представлен фрагмент диаграммы классов телефонной службы приема заявок.



**Рис. 4.3-7.**  Пример диаграммы классов

На этом рисунке показаны основные классы сервера телефонной системы обработки заявок: класс CPBX\_Agent отвечает за работу с PBX (с "железкой", занимающейся коммутацией абонентов); класс COperator содержит описание экземпляров операторов, работающих в call-центре, класс CSubOperator описывает особенного, "продвинутого" оператора (например, начальника над группой операторов); класс COperatorList управляет множеством операторов (добавляет их в список, удаляет и т.д.); класс CNetworkConnectionSupport отвечает за поддержку соединений сервера через локальную компьютерную сеть с компьютерами операторов; класс CDispatсher отвечает за синхронизацию всех остальных программных сущностей на сервере.

Итак, на диаграммах классов изображаются сами классы с атрибутами, типами атрибутов, методами, их параметрами и типами, а так же иерархия наследования классов. И класс, и наследование в UML полностью соответствуют конструкциям объектно-ориентированных языков программирования. Но кроме них на диаграммах классов могут также присутствовать связи между классами - ассоциации. Так, на [рис. 4.3-7](http://www.intuit.ru/department/se/vismodtp/4/#image.4.1) класс CDispatcher связан с классами CPBX\_Agent, COperator, CServerNetworkConnectionSupport и COperatorList.

В UML существует конструкция, которая обобщает класс. Это классификатор (classifier), с помощью которого задаются элементы в модели, которые могут иметь экземпляры, а также операции и атрибуты. Например, экземпляры класса - это объекты. Еще одним примером классификатора является компонента (точнее, тип компоненты). В UML-модели возможны и экземпляры компонент. Кроме класса и компоненты в UML есть и другие классификаторы.

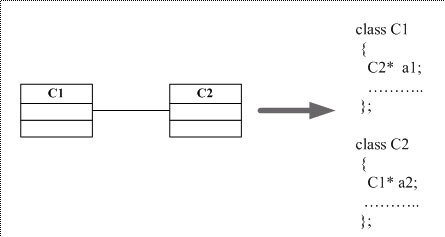
***Ассоциации (association)***

Если два класса связаны друг с другом ассоциацией, то это означает, что их экземпляры (объекты) определенным образом связаны друг с другом, например:

* вызывают методы друг друга,
* работают с общей памятью,
* объекты одного класса являются параметрами методов другого,
* один класс имеет атрибут с типом другого класса (или указателя на него).

Ассоциация как связь между классами обязательно переходит в связь между экземплярами этих классов. Этим она принципиально отличается от наследования. Ниже мы подробно остановимся на отличиях агрегирования и наследования.

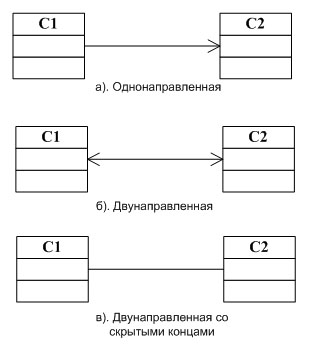
Ассоциации в программном коде могут быть реализованы, например, через атрибуты-указатели языка C++, как показано на [рис. 4.3](http://www.intuit.ru/department/se/vismodtp/4/#image.4.2)-8.



**Рис. 4.3-8.**  Пример представления ассоциаций в программном коде

Отметим, что доступ по ассоциации может быть однонаправленным и двунаправленным. На [рис. 4.3](http://www.intuit.ru/department/se/vismodtp/4/#image.4.3)-9а изображена ассоциация, направленная от класса С1 к классу С2. Это может означать, что в программном коде класса С2 нет атрибута а2, и объекты класса С2 "не знают", что на них ссылаются объекты класса С1. На [рис. 4.3](http://www.intuit.ru/department/se/vismodtp/4/#image.4.3)-9б и в показано два варианта изображения двунаправленной ассоциации. На [рис. 4.3-10](http://www.intuit.ru/department/se/vismodtp/4/#image.4.2) приведен пример реализации именно для двунаправленной ассоциации.

У ассоциации может быть имя, хотя его не обязательно задавать. Имя у ассоциации полезно проставлять, если, например, между двумя классами существует несколько ассоциаций, - чтобы отличать их между собой. Иногда же полезно именовать ассоциации для того, чтобы диаграммы было удобно читать, хотя для этой цели лучше использовать имена их концов, так как они лучше показывают, кто кого использует, обслуживает, включает и пр.



**Рис. 4.3-9.**  Виды ассоциаций

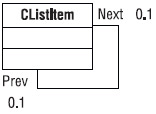
Ассоциации соединяются с классами через специальные конструкции - концы ассоциаций (association ends). Именно у этих концов определяются различные свойства, такие как множественность, агрегирование и др. Концы ассоциаций часто целесообразно именовать на диаграммах, чтобы делать спецификации более наглядными и понятными.

Но именовать концы ассоциаций полезно далеко не всегда. В примере, показанном на [рис. 4.4](http://www.intuit.ru/department/se/vismodtp/4/#image.4.4), концы двух ассоциаций со стороны абонента целесообразно именовать, чтобы станция могла различать абонентов, полученных по разным экземплярам ассоциаций. Противоположные же концы можно не именовать, так как экземпляр класса "Станция" в системе один, у всех других объектов на него имеются ссылки.



**Рис. 4.3-10.**  Пример именования концов ассоциаций

Концы ассоциации имеет смысл именовать всегда, когда ассоциация рефлексивна, то есть связывает класс с ним самим же. Пример такой ассоциации показан на [рис. 4.3-11](http://www.intuit.ru/department/se/vismodtp/4/#image.4.5). Там представлен класс CListItem, реализующий элемент двусвязного списка. У него есть ассоциация с самим собой, которая указывает, что один объект этого класса связан этой ассоциацией с двумя другими объектами - с одним через конец Prev (то есть с предыдущим в списке), с другим - через конец Next (то есть со следующим в списке), а может быть связан только с одним (предыдущим или следующим, и тогда он является, соответственно, последним или первым в списке) или ни с одним вовсе (в этом случае этот объект является единственным в списке). Эти роли используются в качестве имен для концов ассоциаций. Количество объектов, с которыми может быть связан экземпляр класса CListItem по этой ассоциации, указывается с противоположного конца ассоциации с помощью конструкции "множественность".



**Рис. 4.3-11.**  Пример рефлексивной ассоциации

У конца ассоциации есть свойство под названием множественность (multiplicity). Это свойство определяет количество представителей (конкретных объектов), которые могут быть связаны с партнером ассоциации через этот конец. Множественность является целочисленным интервалом или константой. Ниже представлены примеры.

0..1.

1.

0..\*.

1..\*.

\* - просто много, без уточнения того, 0 или 1 фигурируют в качестве нижнего предела.

Константа (например, 2, 10, 100).

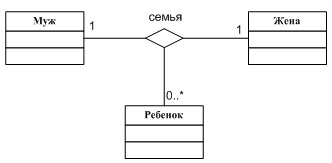
Интервал (например, 3..5, 10..20).

Варианты с первого по пятый являются наиболее распространенными на практике.

Примеры множественности концов ассоциации можно видеть на [рис. 4.3-7](http://www.intuit.ru/department/se/vismodtp/4/#image.4.1) - класс CDispatcher связан с классом COperator так, что каждый экземпляр класса COperator имеет связь ровно с одним экземпляром класса CDispatcher, а каждый экземпляр класса CDispatcher имеет связь с несколькими экземплярами класса COperator или может не иметь такой связи вовсе. Последнее обеспечивается нижней границей множественности ассоциации со стороны класса COperator, равной нулю.

***N-арные ассоциации и класс-ассоциации***

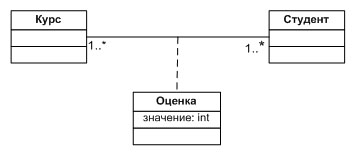
В этом курсе рассматриваются в основном бинарные ассоциации (binary association) - то есть те, которые связывают два класса. Выше были рассмотрены также рефлексивные ассоциации. Вместе с тем в UML возможны n-арные ассоциации (n-ary associations), которые связывают между собой несколько классов. Например, ассоциация под названием "Семья" может связывать следующие классы: "Муж", "Жена", "Ребенок", как показано на [рис. 4.3-12](http://www.intuit.ru/department/se/vismodtp/4/2.html#image.4.6).



**Рис. 4.3-12.**  Пример N-арной ассоциации

В этом примере муж и жена должны присутствовать в семье обязательно (значение множественности у соответствующего конца ассоциации "Cемья" равно 1), а детей может быть произвольное количество, в том числе и не быть вовсе (значение множественности у соответствующего конца ассоциации равно 0..\*).

Ассоциация не может иметь атрибутов, но во многих случаях это крайне желательно. Например, если студент связан ассоциацией "многие-ко-многим" с курсом, то этой ассоциации целесообразно иметь атрибут под названием "оценка". Это достигается связыванием с ассоциацией специального класса - класса-ассоциации (association class), в котором и указываются все нужные атрибуты, как показано на [рис. 4.3-13](http://www.intuit.ru/department/se/vismodtp/4/2.html#image.4.7).



**Рис. 4.3-13.**  Пример класса-ассоциации

***Агрегирование***

У конца ассоциации может быть одно важное свойство под названием агрегирование (aggregation), которое обозначает наличие связи "целое/часть" (part of) между экземплярами классов. Объект-часть в той или или иной форме включается в объект-целое. Так, например, на [рис. 4.3-7](http://www.intuit.ru/department/se/vismodtp/4/1.html#image.4.1) показано, что объекты класса COperator входят в объект класса COperatorList (то есть первый класс агрегируется вторым). Таким образом, агрегирование, как частный случай ассоциации, также обязательно переходит в связи между экземплярами классов.

С помощью агрегирования, например, нельзя определить nested-класс в Java-приложении. Подобный недостаток выразительных средств UML (в частности, для Java) является предметом обширной дискуссии и приводит к созданию диалектов UML, которые являются менее общими, но точнее отражают нужды конкретных платформ реализации. Эти специализации можно создавать, используя встроенные в UML средства - механизм профайлов и extention-механизм. А можно создавать свой собственный визуальный язык и реализовать его, пользуясь DSM-средствами, которых сейчас много - например, Microsoft DSL Tools или Eclipse/GMF. Подробнее об этом будет рассказано в следующих лекциях.

Агрегирование может быть "слабым", как в примере на рис. 4.3-7, и "сильным". В последнем случае оно называется композицией (composition) и означает, что объект-агрегат несет полную ответственность за создание и удаление, а также существование объектов, которые связаны с ним композицией, а также один объект в каждый момент своего существования может одновременно быть частью только одного агрегата. Если в примере с классами COperator и COperatorList последний строго контролирует доступ к каждому оператору из своего списка (создание, удаление, обращение к оператору по номеру в списке и пр.), то можно обозначить связывающую их ассоциацию агрегирования как композицию:

Пример композиции

**Рис. 4.3-14.**  Пример композиции

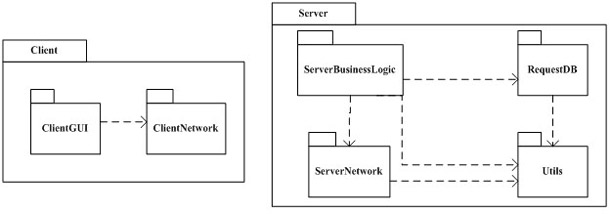
Семантика агрегирования и композиции не определена строго в UML. Отношение "целое-часть" - вот что можно считать определением агрегирования. Ведь на практике существует множество самых разных вариантов такой семантики. Например, "отец" всегда создает и удаляет свои части сам, или только создает, а удалять могут и другие. "Отец" может также поддерживать целостность и корректность своих частей, а может и не заниматься этим. При удалении "отца" "дети" могут удаляться, а могут и нет. "Отец" может брать на себя все взаимодействие своих частей с внешним контекстом (так, что это внешний контекст даже не "видит" его частей) и т. д.

Агрегирование принципиально отличается от наследования. Это важно, поскольку при моделировании предметной области с помощью диаграмм классов UML можно заметить их определенное сходство: (i) оба позволяют строить древообразную иерархию классов; (ii) предок, так же как и агрегируемый класс, добавляет функциональности в потомок/агрегат; (iii) изображения обоих отношений чем-то похожи визуально. И мне как-то раз пришлось в непростом диалоге убеждать аналитиков, что наследование и агрегирование - это не одно и то же. И вот почему.

1. Наследование является только отношением между классами и не переходит в отношение между экземплярами классов, в отличие от агрегирования. Объект, порожденный от класса-наследника, содержит в себе объект класса-предка, но никаких отношений между ними нет – второй есть несамостоятельная часть первого.
2. Наследование модифицирует класс-предок, непосредственно добавляя, "вливая" в него новые свойства (атрибуты, методы, реализацию методов). Агрегирование никак не затрагивает агрегат, и у последнего могут быть свои собственные методы и атрибуты. В случае агрегирования части не "растворяются" в целом, оставаясь отдельными частями в его составе.

***Диаграммы пакетов (package diagrams)***

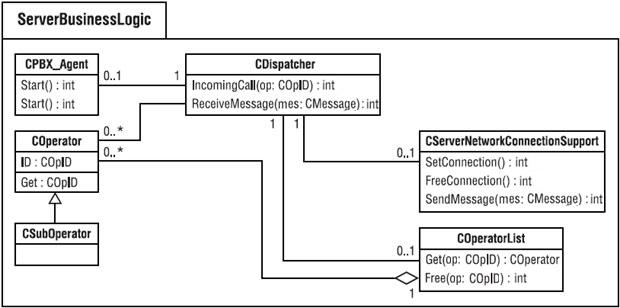
1. Пакет (package) - это конструкция UML, предназначенная для упорядочивания UML-моделей, а также для группировки классов.
2. Пакет, во-первых, выполняет служебную роль, позволяя организовать порядок в создаваемых UML-моделях и распределить различные модельные конструкции, а также диаграммы, по разным "папкам".
3. Во-вторых, в пакеты традиционно помещают классы системы, особенно если проект большой и их много. При этом пакеты UML могут соответствовать, например, проектам (projects) Microsoft Visual Studio. Однако пакеты UML могут быть многократно вложены друг в друга, а проекты Microsoft Visual Studio вложенными быть не могут.
4. В Microsoft Visual Studio есть так называемые рабочие области (solutions), которые содержат в себе проекты. Но на компьютере каждого разработчика проекта могут быть созданы свои собственные рабочие области, содержащие нужные ему проекты, а рабочая область всего приложения, используемая для целостной сборки приложения, может быть вообще другой. Так что рабочие области являются плохими кандидатами на роль пакетов, включающие в себя пакеты-проекты.
5. Пакеты связываются друг с другом специальным отношением - зависимостью (dependence). Это направленное отношение, и идет оно от того пакета, который зависит, к пакету, который необходим тому, первому, зависимому. Это означает, что используемый пакет содержит описание конструкций, которые зависимый пакет импортирует, а не реализует сам. Зависимость не ограничивается только диаграммами пакетов, но может быть использована и для связи других UML-конструкций, например, может связывать два случая использования: один из них может зависеть от другого.
6. Пример диаграммы пакетов изображен на [рис. 4.3-15](http://www.intuit.ru/department/se/vismodtp/4/3.html#image.4.9). Пакет Client содержит два пакета - ClientGUI, в котором находится описание пользовательского интерфейса, и ClientNetwork, отвечающий за сетевое взаимодействие с сервером. При этом первый пакет зависит от второго. В данном случае зависимость означает обычную зависимость проектов в Visual Studio.



**Рис. 4.3-15.**  Пример диаграммы пакетов

Пакет Server содержит все проекты приложения, которые реализуют работу сервера. ServerBusinessLogic содержит весь код, реализующий бизнес-логику сервера, ServerNetwork реализует сетевое сообщение с клиентом, RequestDB - примитивы доступа и логику работы с базой данных запросов. Пакет Util является служебным пакетом где находятся все вспомогательные типы данных, классы, операции и т. д., которые используются всеми пакетами сервера.

На [рис. 4.3-16](http://www.intuit.ru/department/se/vismodtp/4/3.html#image.4.10) средствами диаграмм классов UML показано содержимое пакета ServerBusinessLogic.



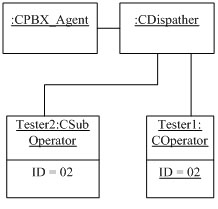
**Рис. 4.3-17.**  Содержимое пакета ServerBusinessLogic в терминах диаграмм классов

В данном случае в пакетах содержится немного классов, да и самих пакетов немного, поскольку в качестве примера представлена упрощенная модель ПО "Телефонной службы приема заявок". В действительности это приложение содержит около пятидесяти различных пакетов и около тысячи классов.

Необходимо отметить, что при проектировании больших приложений, с большим количеством классов и пакетов (в смысле projects в Microsoft Visual Studio), целесообразно создавать диаграммы пакетов. Полезно как-то предварительно прикинуть структуру проектов приложения, хотя, конечно, потом это видение может меняться. Однако ошибки при проектировании структуры пакетов большого приложения приводят к значительным неудобствам, дополнительной работе и к потере концептуальной целостности приложения.

***Диаграммы объектов (object diagrams)***

Этот тип диаграмм предназначен для описания какого-либо фрагмента системы с помощью объектов (objects) - экземпляров классов. Объект является конкретным runtime-экземпляром некоторого класса, имеющим [средство](javascript://) уникальной идентификации, позволяющее отличить его от других объектов того же класса, а также конкретные значения атрибутов и связей. Понятно, что все возможные экземпляры всех классов на диаграммах не изобразить - их слишком много. Поэтому на диаграммы объектов попадают только те экземпляры, которые реально существуют в некотором фрагменте системе в некоторый момент ее работы. Пример такой диаграммы представлен на [рис. 4.3-18](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.11).



**Рис. 4.3-18.**  Пример диаграммы объектов

На этом рисунке изображена следующая конфигурация сервера службы телефонных заявок: один диспетчер (объект ':CDispatcher'), один объект, работающий с PBX (':PBX\_Agent') и два оператора – объекты 'Tester1:COperator' и 'Tester2:CSubOperator'. Для двух первых объектов не указаны имена, поскольку в системе одновременно может быть только по одному такому объекту. Два других объекта соответствуют тестовым операторам, один из которых является "продвинутым" (Tester2, принадлежащий классу CSubOperator).>

Понятно, что информация об экземплярах классов необходима вовсе не для спецификации системы (для этого используются, например, диаграммы классов), а для обсуждения некоторого ее фрагмента. Объект является частным случаем общей концепции экземпляров в UML. Не только классы, но и другие сущности (например, узлы диаграмм развертывания) могут иметь экземпляры.

Общее правило для отображения имен экземпляров таково:

<идентификатор1>: <идентификатор2>,

где <Идентификатор1> - это имя экземпляра, а <Идентификатор2> - имя его классификатора. Строка с именем должна быть подчеркнута.

У объекта может быть также секция атрибутов, где принято указывать значения для атрибутов его класса.

Объект может не иметь имени, как верхние два объекта на [рис. 4.3-18](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.11). Объект также может не иметь класса (или пока не иметь). Наконец, объект может вообще не иметь никакого имени, как и любая конструкция UML. Ведь CASE-пакеты поддерживают идентификацию сущностей UML-моделей по внутренним идентификаторам. Это не очень правильно, но удобно на практике: имена, как и многое другое, можно добавить позже.

Нужно отметить, что изображение имен классов у объектов, как правило, можно отключать, но это не означает, что их нет.

Объекты соединяются друг с другом связями (links). Так же как объекты - это экземпляры классов, так и связи - это экземпляры соответствующих ассоциаций. Однако же в конкретной модели связи могут не иметь ассоциаций. Ведь можно захотеть "накидать" модель объектов просто так, для какого-то документа, обсуждения и пр. И не обязательно строить большую модель с классами. Однако общее определение объектов и связей от этого не страдает - предполагается, что где-то там, за пределами представленных здесь примеров, объектам и связям соответствуют какие-то классы и ассоциации - просто мы не определяем, какие

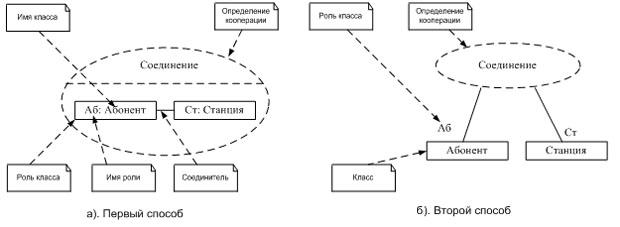
.

***Кооперации (collaborations)***

Так в UML называется описание определенной задачи (например, какой-либо пользовательской функции системы, или внутренней задачи самого ПО, или же какого-либо алгоритма предметной области) в терминах взаимодействующих элементов. Описывается не поведение, а взаимодействующие стороны и их связи. Кооперации показываются на специальном типе диаграмм - на диаграммах композитных структур (composite structure diagrams). Эти диаграммы могут использоваться также для моделирования композитных компонент, как это будет показано в лекциях, посвященных моделированию систем реального времени.

Общающиеся стороны задаются ролями, которые описывают некоторую часть функциональности класса, используемого данным контекстом (в данном случае таким контекстом является кооперация). Например, для двух классов - Абонент и Станция - кооперация "Соединение" (см. [рис. 4.3-19](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.13)) определяет контекст - процедуру установки соединения между абонентом и станцией, а роли этих классов "берут" из самих классов ту функциональность, которая реализует эту процедуру. Ведь кроме этой функциональности в данных классах может быть много разной другой. Роль занимает промежуточное место между классом и его экземплярами[1)](http://www.intuit.ru/department/se/vismodtp/4/footnote.4.1.htm). На [рис. 4.3-20](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.12), а и б представлены примеры двух альтернативных способов задания кооперации.

С этого момента для дальнейшей демонстрации [возможностей](javascript://) UML перестает использоваться пример телефонной службы приема заявок (впрочем, при обсуждении временных диаграмм мы уже отошли от этого примера). Крайне редко бывает так, что при проектировании или описании одной системы используются все типы диаграмм UML. Например, диаграммы классов удобны при проектировании типичного объектно-ориентированного приложения, но при этом редко используются диаграммы состояний и переходов. В то же время последние очень активно применяются при разработке ПО телекоммуникационных систем, совместно с диаграммами композитных структур, а диаграммы классов могут не использоваться вовсе. И так далее. Поэтому, чтобы не запутывать читателей, я, исчерпав данный пример с точки зрения возможностей UML, буду приводить другие примеры, подбирая их наиболее подходящим образом.



**Рис. 4.3-19.**  Способы задания кооперации



**Рис. 4.3-20.**  Пример использования кооперации на диаграмме объектов

Кооперация "Соединение" может быть использована на других диаграммах - на диаграмме объектов, как на [рис. 4.3-20](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.13), а также при определении других коопераций   
(см. [рис. 4.3-21](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.14)). Использование кооперации (collaboration use) - это новая конструкция UML, которая ссылается на определение кооперации и подставляет вместо ее ролей другие роли или объекты, совместимые с ней.

На [рис. 4.3-21](http://www.intuit.ru/department/se/vismodtp/4/4.html#image.4.14) изображено использование кооперации "Соединение" на диаграмме коопераций для создания более сложной кооперации под названием "Соединение абонентов станции". У этой кооперации есть три роли - "Вызывающий абонент", "Вызываемый абонент" и "Своя станция" ("своя" означает, что оба абонента принадлежат одной станции - речь здесь не идет об установлении межстанционного соединения). Эти роли принадлежат классам "Абонент", "Абонент", "Станция" соответственно и подставляются в кооперацию "Станция", образуя два использования этой кооперации - "Исходящее соединение" и "Входящее соединение".



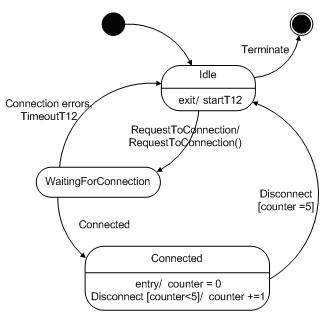
**Рис. 4.3-21.**  Пример использования кооперации при определении другой кооперации

На этом рисунке определяется кооперация "Соединение абонентов одной станции", в которой дважды задействуется кооперация "Соединение": один раз для установки исходящего соединения, другой раз - для входящего. В описании этой кооперации участвуют три роли - "Вызывающий", "Вызываемый" и "Своя станция".

UML требует, чтобы экземпляры классов и роли, которые подставляются как фактические параметры в кооперацию при ее использовании, были совместимы с ее формальными параметрами. Это может означать, что классы подставляемых ролей или экземпляров либо совпадают с классами формальных параметров, либо являются их наследниками.

**Диаграммы конечных автоматов (statechart diagrams)**

На [рис. 4.3-22](http://www.intuit.ru/department/se/vismodtp/4/5.html#image.4.15) приведен пример диаграммы конечных автоматов. Эта диаграмма описывает алгоритм поведения объектов класса COperator системы "Телефонной службы приема заявок", изображенного на [рис. 4.3-7](http://www.intuit.ru/department/se/vismodtp/4/1.html#image.4.1).



**Рис. 4.3-22.**  Пример диаграммы конечных автоматов

После инициализации объекта он переходит в состояние Idle. В этом состоянии объект пребывает, пока свободен и не участвует в приеме заявки от клиента. Когда приходит запрос от клиента, объект переходит в состояние WaitingForConnection - ожидание установки соединения по локальной сети с соответствующим оператором. После получения сигнала Connected объект переходит в состояние Connected, и это означает, что оператор готов работать с данным клиентом. Вся работа оператора с клиентом происходит в этом состоянии.

Из состояния WaitingForConnection объект может перейти в состояние Idle, если ожидание соединения с оператором превысит время T12.

При получении сигнала Disconnect, свидетельствующего об окончании обслуживания клиента, объект переходит в состояние Idle, в котором ожидает новый запрос. В этом же состоянии объект может обработать сигнал Terminate - указание завершить всю свою работу и освободить оперативную память.

В состоянии Connected объект может получить четыре сигнала Disconnect, не реагируя на них, но при получении пятого он переходит в состояние Idle.

Подробно этот тип диаграмм UML будет рассмотрен в лекциях, посвященных моделированию систем реального времени.

**4.3.4.Контрольные вопросы**

1. Перечислите и кратко охарактеризуйте типы диаграмм UML.
2. Сколь строго разделение UML на типы диаграмм?
3. Что такое актер? Перечислите типы актеров и расскажите об их особенностях.
4. Какие отношения возможны между актерами?
5. Что такое случай использования? Расскажите о критериях создания случев использования.
6. Что такое диаграммы бизнес-случаев использования и зачем они нужны?
7. Расскажите, для чего, на ваш взгляд, нужны диаграммы случаев использования.
8. Расскажите о предназначении диаграмм активностей.
9. Расскажите о структуре диаграмм активностей.
10. Чем параллельный разветвитель отличается от логичеcкого?
11. Какой аспект системы призваны моделировать диаграммы развертывания?
12. Каких видов бывают диаграммы развертывания?
13. Какие виды узлов смогут присутствовать на диаграмме развертывания?
14. Предназначаются ли диаграммы развертывания для полной спецификации аппаратной части системы?
15. Расскажите о вариантах использования диаграмм развертывания.
16. Что такое компонента ПО?
17. В чем может выражаться независимоcть компонент?
18. Что такое интерфейс компоненты?
19. Расскажите о проблеме поддержания UML-диаграмм проекта в актуальном состоянии.
20. Как диаграммы компонент могут быть связаны с диаграммами развертывания? Приведите собственный пример.
21. Чем похожи и в чем различаются диаграммы последовательностей и коммуникаций? Какие из них, на ваш взгляд, ближе к структурным, а какие - к поведенческим?
22. Расскажите, в каких случаях, на ваш взгляд, целесообразно применять временные диаграммы. Что в них есть такого, что отсутствует в других поведенческих диаграммах UML?
23. Приведите примеры ситуаций в разработке ПО, когда полезно использовать диаграммы коммуникаций и последовательностей.
24. На примере использования диаграмм последовательностей, изложенном в лекции, постарайтесь обосновать полезность предварительного проектирования ПО.
25. Расскажите о диаграммах схем взаимодействия.
26. Попробуйте составить несколько диаграмм последовательностей и связать их вместе, используя диаграммы схем взаимодействия.
27. Какую информацию о разрабатываемой системе принято изображать на диаграммах классов?
28. Дайте определение ассоциации. Чем она отличается от наследования?
29. Какие отношения между классами не переходят в связи между экземплярами?
30. Что такое конец ассоциации?
31. Чем однонаправленная ассоциация отличается от двунаправленной?
32. Приведите свой пример n-арной ассоциации.
33. Расскажите о соображениях по именованию концов ассоциаций. Приведите примеры для бинарных ассоциаций, когда именование концов облегчает чтение диаграммы, и пример, когда это оказывается избыточным.
34. Как, на ваш взгляд, сочетаются имя ассоциации и имена ее концов на одной диаграмме? Приведите собственный пример для подтверждения своего мнения.
35. Зачем нужны классы-ассоциации? Постройте собственный пример.
36. Дайте определение агрегирования. Приведите примеры различных семантик агрегирования.
37. Агрегирование является: свойством ассоциации, свойством роли, отдельным отношением между классами UML?
38. Может ли у двух и более концов ассоциации быть свойство агрегирования?
39. Возможна ли ситуация, когда класс агрегируется несколькими другими классами, но тем не менее любой его экземпляр агрегируется только одним объектом? Приведите пример.
40. Может ли класс агрегироваться несколькими другими классами, и при этом его экземпляр также будет входить в несколько объектов? Верно ли то же утверждение про композицию? Приведите содержательный пример.
41. Чем агрегирование похоже на наследование и чем отличается?
42. Что такое композиция? Приведите пример.
43. Попробуйте расширить UML для выражения связи, которая существует между классом и вложенным в него классом (nested-класс языка Java).
44. Что такое UML-пакет?
45. Чем пакеты UML близки к проектам и solutions Microsoft Visual Studio?
46. Могут ли пакеты содержать элементы UML-модели, отличные от других пакетов и классов?
47. Что такое зависимость между пакетами? Может ли это отношение использоваться для других UML-элементов?
48. Дайте определение объекта.
49. Расскажите о правилах изображения имен у сущностей UML, соответствующих каким-либо экземплярам (в частности, объектам классов).
50. Что такое связи между объектами?
51. Что такое кооперация? Приведите свой пример.
52. Приведите примеры альтернативных определений кооперации.
53. На каких диаграммах может использоваться кооперация? Приведите собственные примеры.
54. Что такое совместимость фактических и формальных параметров кооперации?
55. Расскажите о правилах изображения имен ролей.
56. Что такое диаграммы конечных автоматов? Приведите свой собственный пример.

**4.3.5. Тестовые задания**