# Лекция № 3. Введение в UML 2.0.

*В этой лекции делается обзор всех типов диаграмм UML 2.0. Этот обзор делается на основе одного главного примера – телефонной службы приема заявок.*

**Диаграммы UML.** Настала пора познакомиться с одним из языков визуального моделирования. Для этой цели выберем UML (Unified Modeling Language) версии 2.0, являющийся на данный момент самым распространенным языком визуального моделирования программного обеспечения.

Мы предпримем лишь краткий экскурс в UML, поскольку, с одной стороны, существует большое количество материалов по этой теме, а, с другой стороны, как-то неприлично обсуждать визуальное моделирование, минуя UML. Мы будем рассматривать различные виды диаграмм UML, позволяющие строить модели ПО с разных точек зрения.

Виды диаграмм UML являются нотацией (конкретным синтаксисом) языка. Абстрактный синтаксис и семантика определяются с помощью метамодели UML. Вообще говоря, типы диаграмм не являются строго обязательными при использовании UML – различные конструкции языка можно вставлять в разные диаграммы. Они, виды диаграмм – это всего лишь наиболее устоявшиеся способы использовать UML.

Итак, авторы UML выделяют следующие виды диаграмм (см. рис 3.1):

- Структурные диаграммы:
  - диаграммы классов (class diagrams);
  - диаграммы компонент (component diagrams);
  - диаграммы объектов (object diagrams);
  - диаграммы композитных структур (composite structure diagrams);
  - диаграммы развертывания (deployment diagrams);
  - диаграммы пакетов (package diagrams).

- Поведенческие диаграммы:
  - диаграммы активностей (activity diagrams);
  - диаграммы случаев использования (use case diagrams);
  - диаграммы конечных автоматов (state machine diagram);
  - диаграммы взаимодействий (interaction diagram):
    - диаграммы последовательностей (sequence diagram);
    - диаграммы схем взаимодействия (interaction overview diagram);
    - диаграммы коммуникаций (communication diagrams);
    - временные диаграммы (timing diagrams).

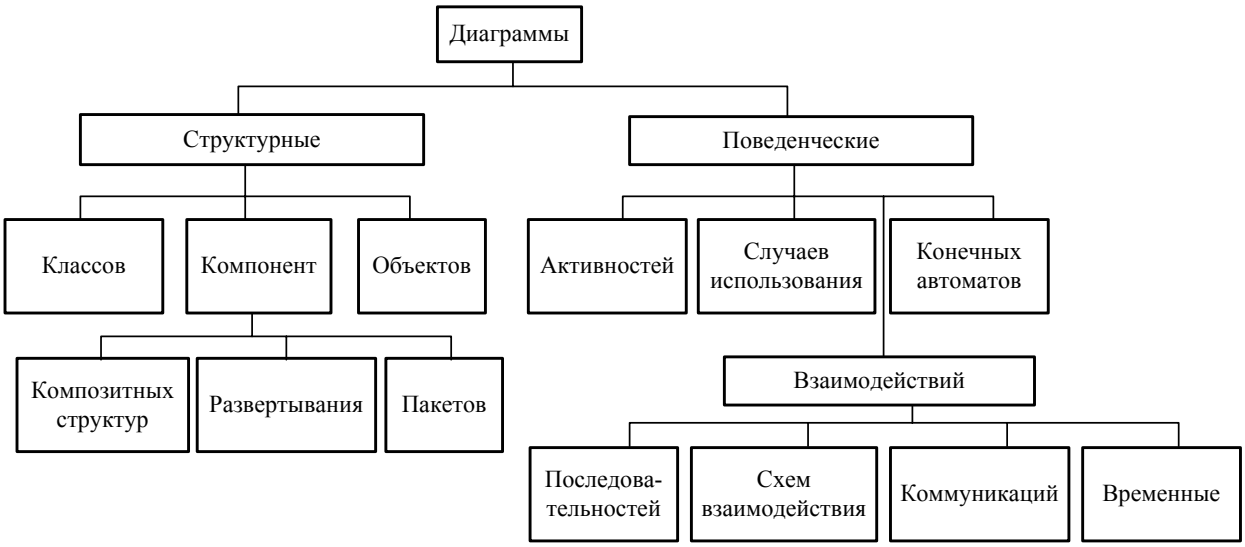



Рис. 3.1.

Необходимо отметить, что разбиение нотации UML на диаграммы весьма условно. Вообще, произвольные конструкции UML могут присутствовать на одной диаграмме, одну тип конструкций может раскрывать другой. Например, на рис. 3.15 показано, как на диаграмме классов могут присутствовать пакеты, хотя диаграммы классов и диаграммы пакетов – разные виды диаграмм. Диаграммы, перечисленные выше, соответствуют не пересекающемуся покрытию метамодели UML, используемому при его описании в самом стандарте. Можно сказать и так, что эти диаграммы соответствует принятым точкам зрения на ПО, широко используемым при моделировании.

**Система “Телефонная служба приема заявок”.** Мы продолжим знакомство с UML на примере разработки системы “Телефонная служба приема заявок”. Мы не просто приведем фрагменты этой системы, изображенные с помощью различных UML-диаграмм, но и снабдим диаграммы некоторым связным сюжетом, дающим понять, как эти диаграммы могли появиться на свет. Разумеется, это сюжет далеко не единственный, даже в рамках разработки данной системы. Он нужен, чтобы с первых же шагов при знакомстве с UML не появлялось чувство, что пустоты, подвешенных в воздух примеров.

Итак, заказчик нашей системы – это компания, владеющая сетью продуктовых магазинов. Данная компания, кроме обычной розничной торговли, хочет предоставлять еще также и сервис по обслуживанию клиентов по телефонным заявкам. 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, и заказываемая система должна быть с ней проинтегрирована.

Основная трудность в реализации этой задачи – это подходящее программное обеспечение. Компания делает заказ на разработку соответствующей программной системы.

**Диаграммы случаев использования** (use case diagrams). Первым шагом по реализации описанной выше задачи является уточнение требований. Для этого можно использовать диаграммы случаев использования UML. Пример такой диаграммы для нашей системы представлен на рис. 3.2.

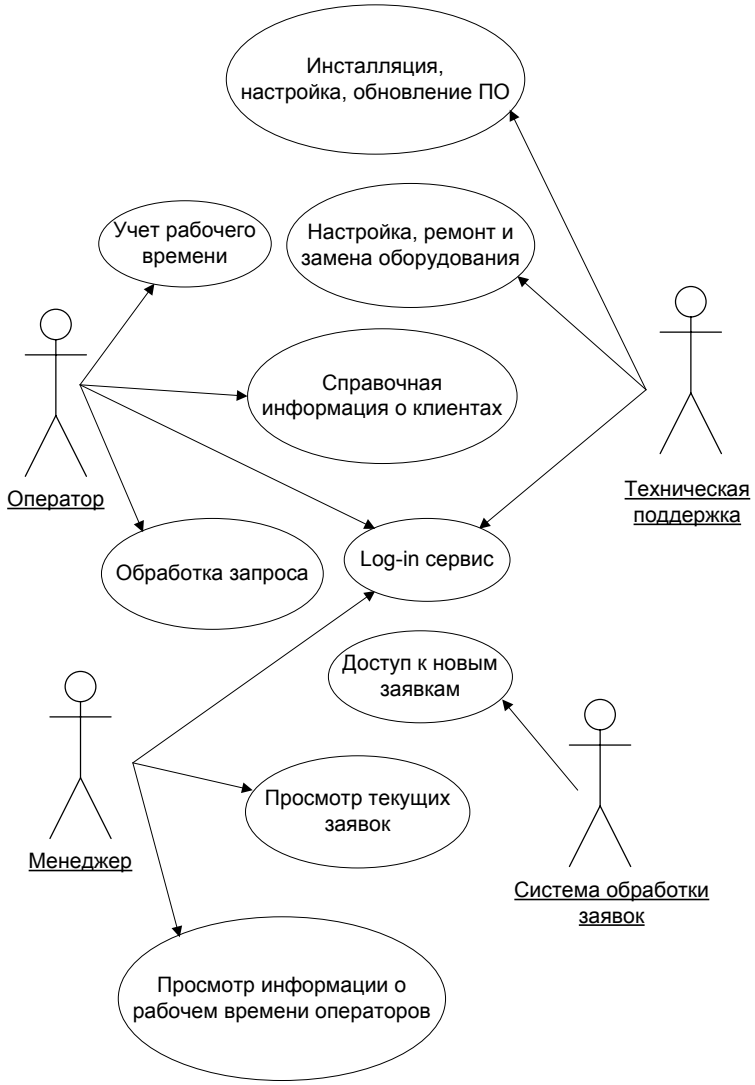



Рис. 3.2.

На ней мы обозначили три вида пользователей – оператора, менеджера и представителей технической поддержки. Система должна также поддерживать внешний интерфейс системой обработки заявок. Это – четвертый пользователь, или, в терминах UML, актер. Случаи использования системы комментировать не будем, считая, что итак все понятно из картинки.

Случаи использования – это полный, но не подробный список сервисов системы, который она должна предоставить пользователю. Реализацию системы удобно проводить в терминах этих случаев использования. Казалось бы, что проще – реализовать набор функций, необходимых пользователю. Однако на деле программный проект может незаметно потерять свою главную цель – реализацию нужной для пользователя функциональности. Вместо этого (то есть реализации заказанной функциональности) можно, например, очень долго заниматься реализацией 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 (то есть проект весьма сильно выбился из расписания). Кроме того, 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 требования к ней внезапно и сильно меняются, или когда на ее основе делаются другие системы. Необходим баланс между внутренним совершенством программного обеспечения и функциональность заказчика, доставленной ему в срок. Вести разработку именно в терминах случаев использования является хорошим способом контролировать, что мы двигаемся в нужном направлении.

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 использоваться обычный текст (по одному абзацу на каждый случай использования), и/или другие диаграммы UML. Более подробно об использовании этого типа диаграмм UML мы поговорим при обсуждении методологии RUP/USDP.

**Диаграммы активностей** (activity diagrams). С их помощью удобно изображать бизнес-процессы – алгоритмы, по которым работает компания. Именно в эти алгоритмы должна встроиться наша информационная система, автоматизировав часть ее функций. В нашем случае в компании должен быть создан новый бизнес-процесс по телефонной обработке заявок. Заказчик как-то себе представляет этот будущий процесс. Перед началом разработки системы необходимо уточнить алгоритм работы этой новой службы. На рис. 3.3 показана общая схема работы оператора с клиентом.

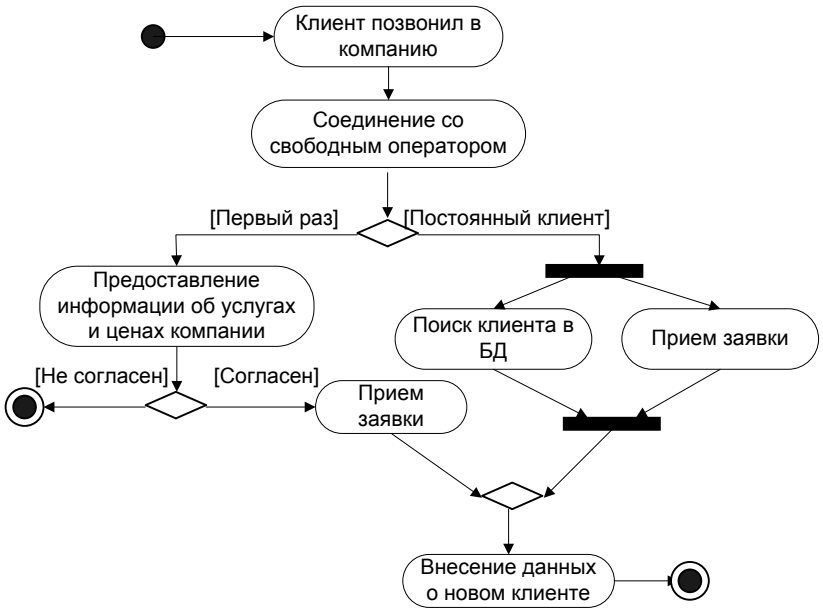


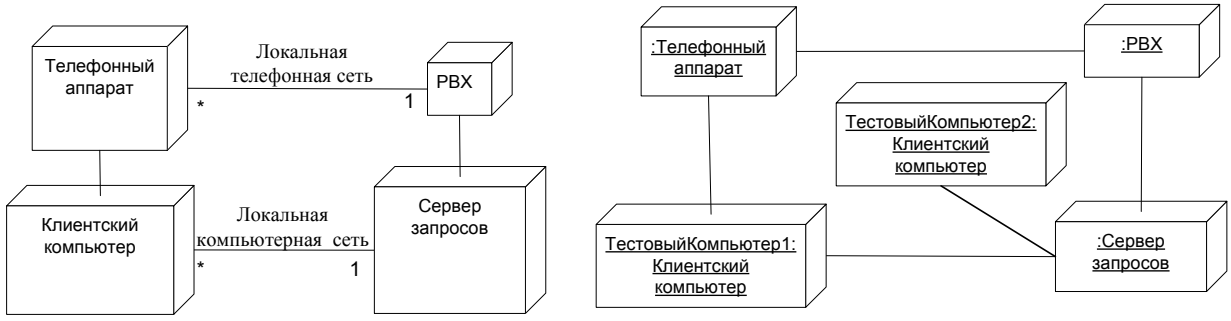

Рис. 3.3.

Данная диаграмма является не полной по следующим причинам. Многие узла, например, "прием заявки", могут оказаться достаточно сложными – необходимо узнать у клиента перечень продукции, которая ему нужна, сосчитать и назвать ему цену,

возможно, изменить после этого список (если цена окажется слишком большой), согласовать с ним дату и время доставки и т.д.

Все бизнес-процессы компании, в которые прямо или косвенно будет вовлечена новая система, должны ясно представляться программистами. В нашем случае у компании еще есть бизнес-процесс обработки заявок, который уже работает есть у заказчика, и его также нужно понять. Иначе может оказаться, что упущена какая-то важная деталь, которая не позволяет новой системе полновесно выполнять свои функции. Например, может оказаться, что подсистема обработки заявок реализована … на макросах к Word/Excel и не готова обрабатывать объем информации, который предоставит ей наша система. На этот факт необходимо указать заказчику, так как иначе проект не закончиться успешно – заказчик потратит деньги и не получит новой функциональности в свой бизнес. По этой причине он может больше не обратиться к этой же фирме-исполнителю.

**Диаграммы развертывания (deployment diagrams).** Теперь настало время в первом приближении определить будущую систему изнутри. На рис. 3.4 (а) показано, что система будет состоять из офисной телефонной станции (PBX – Public Branch Exchange), сервера, телефонных аппаратов и клиентских компьютеров. На этом рисунке представлена диаграмма развертывания в одном из двух возможных в UML видов – в описательном. То есть на ней определены типы узлов системы, а между ними – ассоциации с пометками множественности (см. описание диаграмм классов). На рис. 3.4 (б) представлена диаграмма развертывания в экземпляром варианте. Показан тестовый вариант системы, который, кроме сервера и PBX, содержит один пользовательский компьютер для тестирования взаимодействия сервера и клиента, и один клиентский компьютер вместе с телефонным аппаратом для тестирования связи клиента с сервером и PBX. Два клиентских компьютера нужны, чтобы тестировать работу ПО в случае более чем одного клиента (при переходе от одного к двум начинают появляться многочисленные ситуации, которые не проявлялись ранее. Большее количество клиентов – 3,10 и т.д. – может быть не принципиально на первых стадиях тестирования и отладки. Эти виды диаграмм развертывания соотносятся между собой, как диаграммы классов и диаграммы объектов.




(а) Диаграмма развертывания: описательный уровень

(б) Диаграмма развертывания: экземплярный уровень

Рис. 3.4.

Такая диаграмма может понадобиться, например, как приложение к техническому заданию, при обсуждении цен на различные офисные АТС, телефонные аппараты и компьютеры. Наконец, эту диаграмму может нарисовать менеджер проекта перед тем, как начать обсуждение с разработчиками архитектуры системы. Эта диаграмма может лежать на столе во время первых таких обсуждений, пока не родилось ничего более конкретного, что также можно нарисовать. Она может оказаться полезной при общении технических

специалистов с нетехническими людьми – вся система здесь, фактически изображена, но с минимум технических деталей.

**Диаграммы компонент (component diagrams).** В качестве следующего промежуточного результата обсуждений архитектуры может появиться диаграмма, приведенная на рис. 3.5.

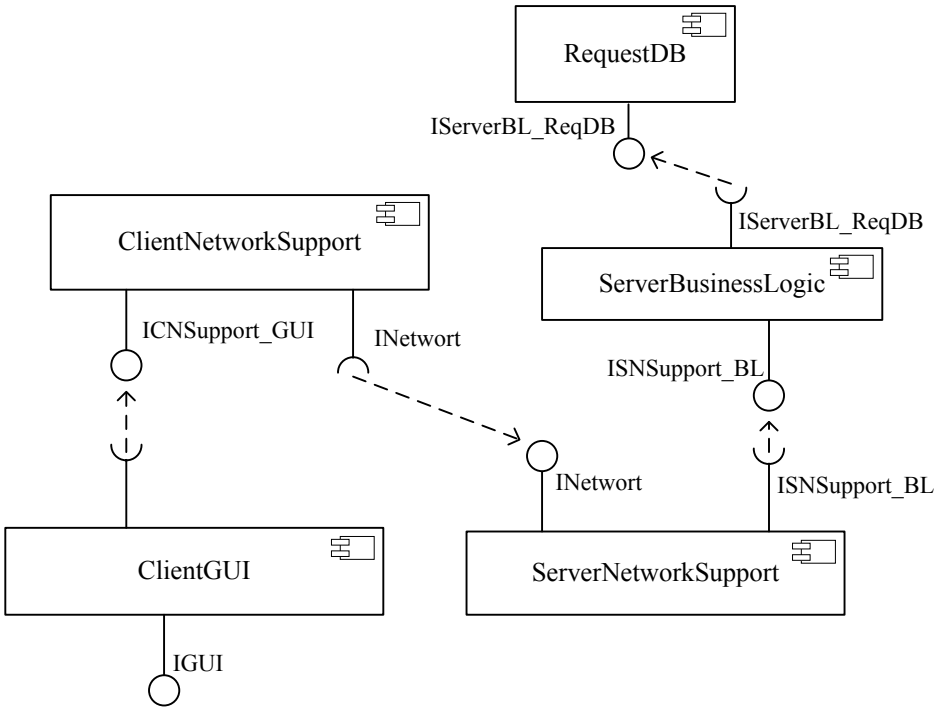



Рис. 3.5.

Это – диаграмма компонент UML, на которой изображены независимые модули нашего ПО, взаимодействующие друг с другом через интерфейсы. Независимость выражается, во-первых, в том, что эти модули реализуют существенно различную функциональность. Модуль ClientGUI реализует пользовательский интерфейс рабочего места оператора, модули ClientNetworkSupport и ServerNetworkSupport реализуют поддержку сетевого взаимодействия между клиентом и сервером, модуль ServerBusinessLogic реализует бизнес-логику сервера, а модуль RequestDB отвечает за взаимодействие с базой данных заявок и синхронизацию с системой обработки заявок. Во-вторых, разработку каждого такого модуля можно поручить отдельному разработчику или команде разработчиков, то есть осуществить организационное разделение коллектива программистов. В-третьих, каждый такой модуль независим с точки физической организации – то есть представляется в виде набора независимых бинарных файлов (например, dll-файлов). В-четвертых, возможна также независимость периода исполнения - каждый из модулей может находиться или на отдельном компьютере, или в отдельном процессе операционной системы, или существовать в виде набора отдельных нитей (threads).

В силу независимости, а также необходимости взаимодействия, эти модули имеют интерфейсы, скрывающие их внутреннее устройство и предоставляющие во вне определенный способ обращения к своим функциям. Будем называть такие модули компонентами. Интерфейс изображается маленьким кружочком, который соединен обычной линией с компонентой, которая его реализует, и пунктирной линией с компонентой, которая его использует.

Отметим, что эта диаграмма может со временем меняться – интерфейсы могут уточняться, могут быть добавлены новые компоненты, существующие могут разбиваться на более мелкие и т.д. При всех этих вариантах продолжения разработки нашего ПО мы

можем поддерживать эту диаграмму, либо она так и останется такой, какой мы ее нарисовали при начальных обсуждениях архитектуры. Отметим, что поддержка этой диаграммы – не простое дело. Как правило, в начале все просто и концептуально, потом – все разваливается на большое число деталей и сроки поджимают – нужно, чтобы работало. Вследствие этого целостность архитектуры ПО уходит из фокуса внимания разработчиков. Однако есть такое правило – если нельзя ясно и кратко выразить главное в какой-либо деятельности, значит там не все порядке. Таким образом, имеет смысл стремиться поддерживать актуальность именно этой диаграммы, поскольку она является одной из основных спецификаций архитектуры нашего приложения.

Еще один важный аспект системы, изображенный на этой диаграмме – интерфейсы компонент. Их нужно прорабатывать особенно тщательно, поскольку если они не согласованы, а клиент и пользователь разрабатываются разными группами, то согласование интерфейсов может потребовать серьезных модификаций в уже написанном коде.

Приведем теперь диаграмму, показывающую, каким образом компоненты нашей системы распределяются по аппаратной части нашей системы – рис. 3.6.

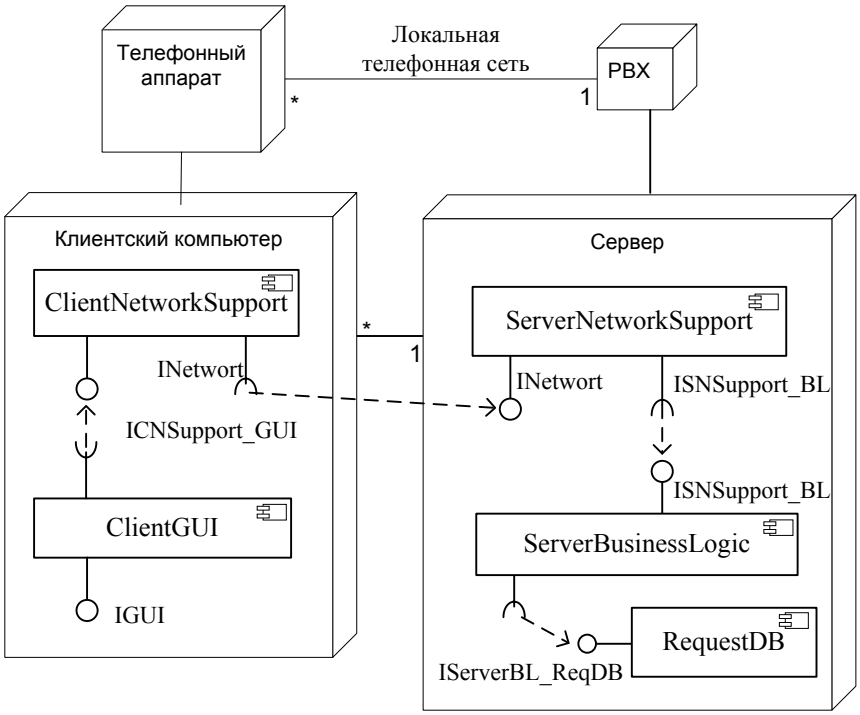



Рис. 3.6.

Отметим, что описание узлов диаграмм развертывания производится на описательном, а не экземплярном уровне.

Однако нужно заметить, что именно предыдущая диаграмма является кандидатом в долгожители в процессе разработки, поскольку лаконична и не содержит лишней информации. То, какие именно компоненты располагаются на сервере, а какие на клиенте – не очень важная деталь здесь, поскольку система не очень большая, все это и так помнят. Кроме того, факт распределения компонент по аппаратуре не является здесь предметом изменений, как в более сложной системе, где существует несколько разных серверов, клиенты различных типов и т.д. Такая диаграмма является, скорее, разовой и полезна для какого-либо отчета, для разговоров с заказчиком и т.д.

**Диаграммы коммуникаций (communication diagrams).** Продолжая разговор об отдельных деталей работы будущей системы, начатый в предыдущем разделе, обратимся к временным свойствам алгоритмов. 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 нашей команды или рабочими группами. Побудительные причины выяснения этих деталей могут быть различными – разработчики одной из компонент вдруг обнаруживают, что они не понимают того контекста, в котором будет работать их компонента. Или тестеры находят ошибки, относительно которых автор каждый компонент утверждает, что его компонента в этой ситуации отработала правильно. Во всех этих ситуациях целесообразно собрать совещание с присутствием всех заинтересованных сторон. При этом самый заинтересованный – тестер, 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 диаграммы. В данном случае мы рекомендуем использовать диаграммы кооперацией.

На рис. 3.7 изображается, как выглядит ситуация поступления в систему звонка от клиента. Эта диаграмма может быть полезной, если мы хотим определить, как информация о звонке распространяется по нашему ПО, какие процессы при этом происходят в его различных частях и какие данные передаются.

Мы видим, что от клиента звонок приходит на офисную АТС, оттуда уходит на телефонный аппарат свободного оператора и на сервер нашего ПО. 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 о том 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 диалога для внесения информации о звонке. Отметим, что вся эта дополнительная информация впрямую на диаграмме не содержится (например, у сообщения номер 3 от PBX к серверной компоненте “Бизнес-логика” нет параметров – информации о проключеном операторе). Эта информация выводится данной в процессе анализа и обсуждения, но ни диаграммы, а проблем и вопросов, частично визуализированных на этой диаграмме.

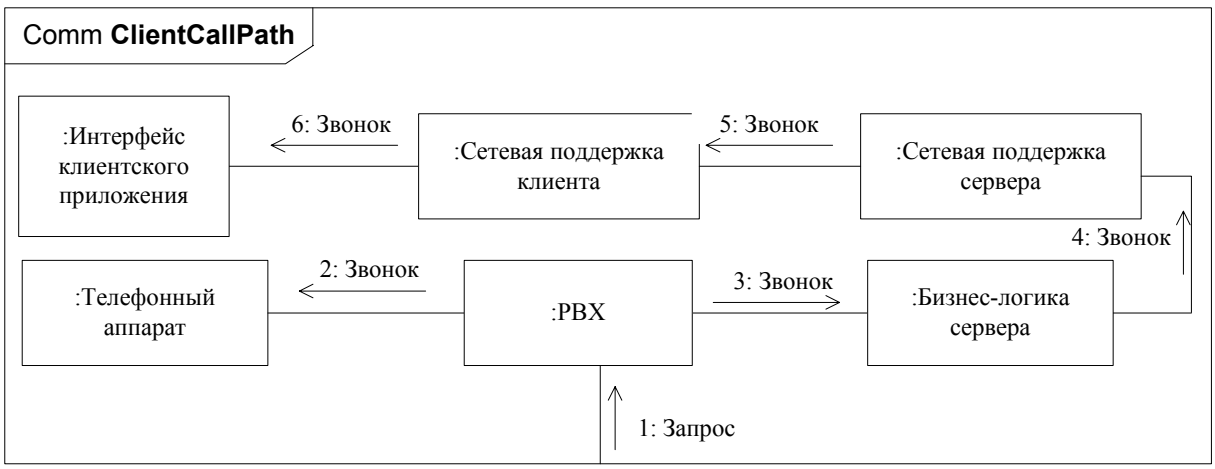



Рис. 3.7.

**Диаграммы последовательности (sequence diagrams).** Продолжая разговор об отдельных деталей работы будущей системы, начатый в предыдущем разделе, обратимся к временным свойствам. Для этого в UML есть диаграммы последовательности. Одна такая диаграмма представлена на рис. 3.8.

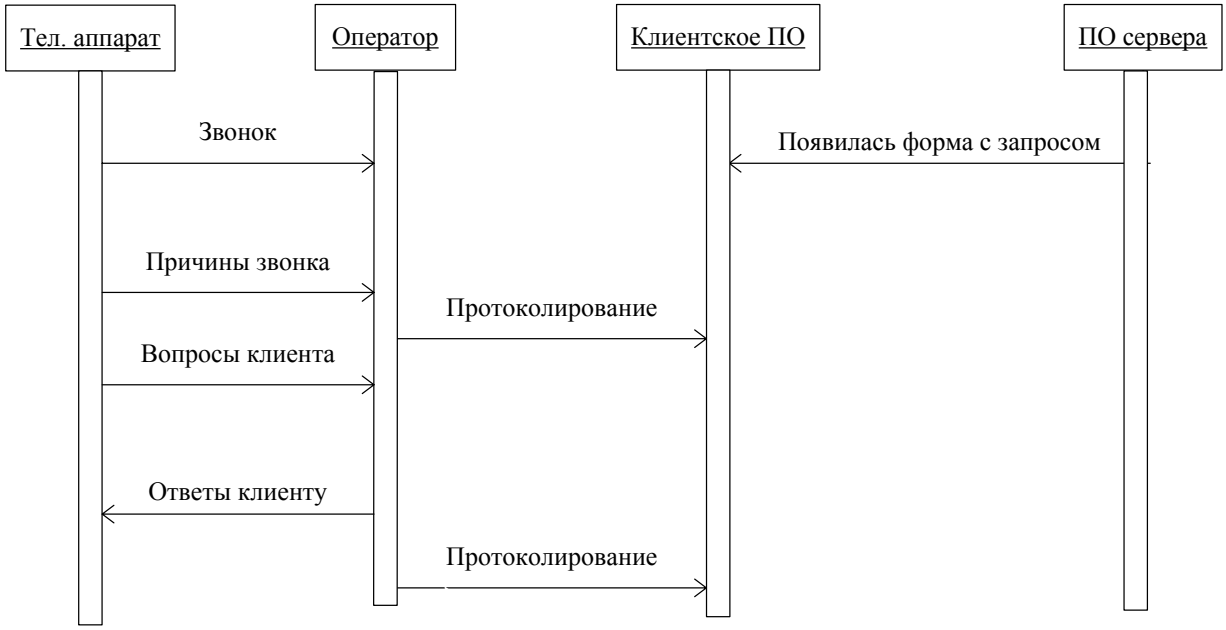



Рис. 3.8.

Эта диаграмма фокусируется на действиях оператора в рамках клиентского ПО. Во-первых, на ней явно изображено, что звонок оператору по телефону и диалог для внесения информации о звонке на дисплее оператора появляются одновременно. Это "одновременно" может впоследствии доставить много хлопот, поскольку необходимо будет тестировать этот требование в условиях, идентичных условиям заказчика – его локальная сеть с тем быстродействием, которое она может обеспечивать, с определенным количеством одновременно работающих в сети операторов и т.д. И понятно, что в этой ситуации наше ПО должно соревноваться по скорости с процессом коммутации в PBX. Вполне возможно, что телефонный аппарат будет звонить существенно раньше, чем соответствующая форма о звонке – появляться на экране оператора, что может оказаться весьма неудобным. То есть нужно «убыстрять» обработку звонка сервером. 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, 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 как и так все понятно, но предварительное обдумывание с фиксацией решений с помощью диаграмм, обсуждение этих диаграмм с коллегами, может предотвратить ошибки, которые, будучи допущенными, потребует существенных больших усилий на исправление, много превышающих те, что были потрачены на проектирование.

**Диаграммы классов (class diagrams).** Этот вид диаграмм является основным при объектно-ориентированной разработке 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 изучении ПО. По этим диаграммам легко генерировать программный код, а также эти диаграммы легко восстанавливать с помощью автоматического возвратного проектирования из уже существующего кода. Кроме того, диаграммы классов используются при описании самих языков визуального моделирования, как это будет показано ниже.

На рис. 3.9. представлен фрагмент модели классов для нашего примера.

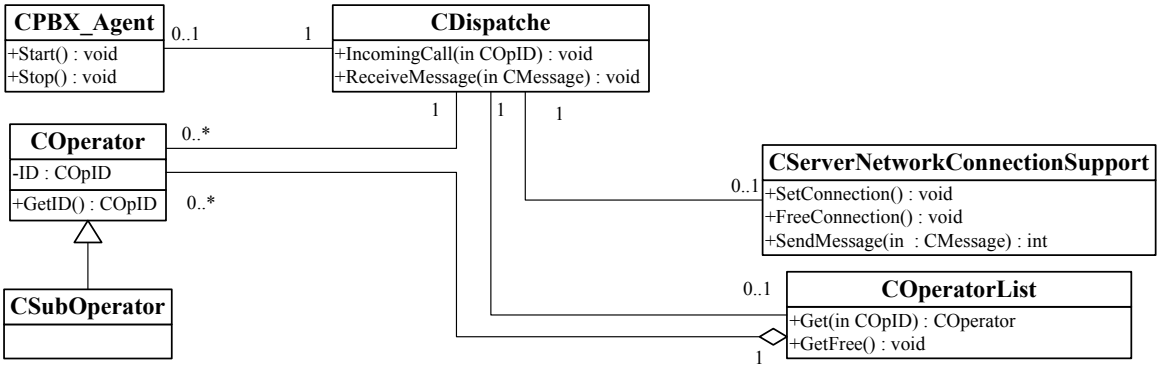



Рис. 3.9

Итак, на диаграммах классов изображаются сами классы с атрибутами, типами атрибутов, методами, их параметрами и типами. Показывается также иерархия наследования классов. Все эти элементы диаграммы классов один в один соответствуют конструкциям объектно-ориентированных языков программирования. Но кроме них на диаграммах классов могут быть также ассоциации. Так на рис. 3.9 класс CDispatcher связан с классами CPBX_Agent, COperator, CServerNetworkConnectionSupport и COperatorList. Поговорим немного про ассоциации.

Тот факт, что два класса связаны друг с другом ассоциацией, означает, что их экземпляры (объекты) определенным образом связаны друг с другом – например, вызывают методы друг друга, работают с общей памятью и т.д. Кроме самого факта наличия связей, ассоциация обеспечивает доступ друг к другу соединенных объектов. Ассоциации в программном коде могут быть представлены, например, через тип атрибута класса, например, у класса C1 есть атрибут a1, который имеет тип C2 – см. рис 3.10.

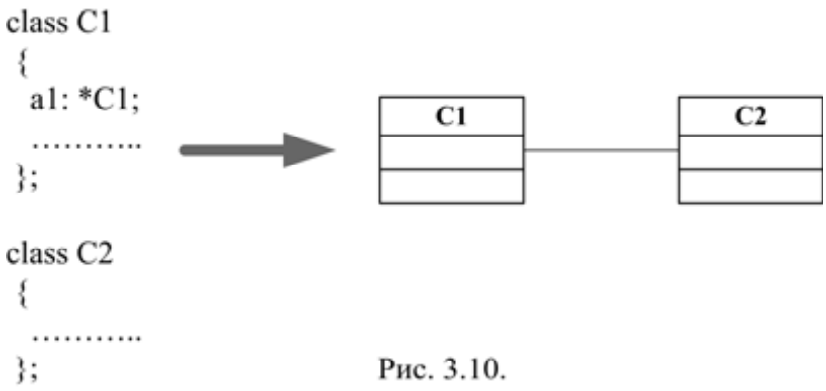



Рис. 3.10.

Отметим, что доступ по ассоциации может быть однонаправленным и двунаправленным. В первом примере, если объект класса C2 не знает, кто его создал (то

есть не имеет указателя на объект-создатель), то ассоциация направлена от C1 к C2 (см. рис. 3.11 (а)), если знает, то ассоциация двунаправленная – см. рис. 3.11 (б). Двунаправленные ассоциации часто показывают без стрелок на концах, как показано на рис 3.11 (в). Отметим также, что у ассоциации может быть имя.

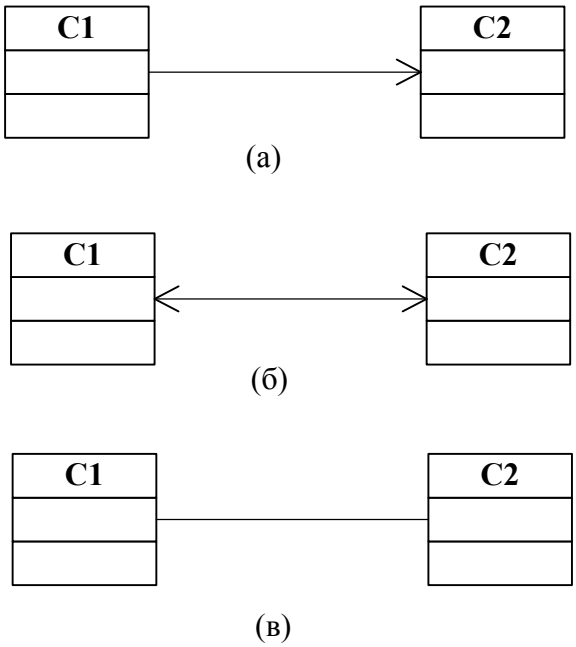



Рис. 3.11.

Ассоциация, связывая два класса, порождает две роли этих классов. Роль класса – это определенный аспект его функциональности. Если роль определена ассоциацией, то этот аспект поведения класса проявляется при взаимодействии его объектов с объектами другого класса в соответствии с данной ассоциацией. То есть ассоциация вовлекает объекты данного класса в определенное взаимодействие с другими объектами и тем самым навязывает классу определенную роль, определяемую ролью его объектов в этом взаимодействии.

Например, на рис. 3.12 представлен класс CListItem, реализующий элемент двусвязного списка. У него есть ассоциация с самим собой указывающая, что один объект этого класса связан этой ассоциацией с двумя другими объектами – с одним в роли Prev (то есть предыдущий для первого объекта), с другим – в роли Next (то есть следующий для второго объекта), а может быть связан только с одни (предыдущим или следующим и тогда он является последним или первым в списке соответственно) или не с одним вовсе (в этом случае этот объект является единственным в списке). Количество объектов, с которыми может быть связан экземпляр класса CListItem по этой ассоциации, указывается с противоположного конца ассоциации с помощью конструкции множественность, о которой чуть ниже. А теперь отметим, какая функциональность класса CListItem высвечивается благодаря ролям Prev и Next.

Это, прежде всего, атрибуты Prev и Next, в которых хранятся указатели на предыдущий и следующий элементы списка. И это проверка того, является ли данный элемент первым/последним в списке.

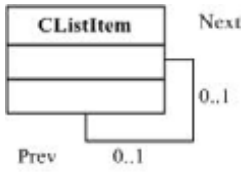



Рис. 3.12.

У роли ассоциации есть свойство под названием множественность (multiplicity), которое определяет количество представителей (конкретных объектов), которые могут быть связаны с партнером ассоциации через данную роль. Множественность является целочисленным интервалом, например:

0..1

10

0..*

3..5,10..20,100,200..*

Так, на рис. 3.9 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 не иметь такой связи вовсе. Последнее обеспечивается нижней границей множественности ассоциации со стороны класса COperator, равной 0.

Наконец, у роли может быть свойство под названием «агрегирование». Агрегирование обозначает наличие связи между объектами типа целое/часть (part of), то есть объект агрегируемого класса в той или иной форме является частью агрегата. Так, например, на рис. 3.9 показано, что класс COperator агрегируется классом COperatorList. Агрегация может быть слабой, как в этом примере, так и сильной. В последнем случае она называется композицией и означает, что объект-агрегат несет полную ответственность за создание и удаление, а также существование объектов, которые связаны с ним по сильному агрегированию. В частности, последние не могут быть связаны агрегированием ни с какими другими объектами. Если в примере с классами COperator и COperatorList последний строго обеспечивает весь 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:

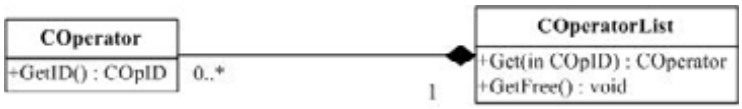



Рис. 3.13.

**Диаграммы пакетов (package diagrams).** Пакет в UML – это контейнер самого общего вида, в который можно сложить какую-либо часть создаваемой модели. Это, так сказать, служебная роль пакета, позволяющая организовать порядок в создаваемых UML-моделях. Но кроме того, в пакеты обычно помещают классы системы, особенно, если их много и проект большой. При этом пакеты UML могут соответствовать, например, проектам (projects) Microsoft Visual Studio. Но – только листовые: ведь пакеты могут быть многократно вложены в друг друга, а проекты Microsoft Visual Studio вложенными быть не могут.

В Microsoft .Net Studio есть так называемые рабочие области (solutions), которые содержат в себе проекты. Но на компьютере каждого разработчика могут быть созданы свои собственные рабочие области, содержащие нужные ему проекты, а рабочая область всего приложения, которая используемая для целостной сборки всего приложения, может быть вообще другой. Так что рабочие области являются плохими кандидатами на пакеты, включающие в себя пакеты-проекты.

Пример диаграммы пакетов для нашей системы изображен на рис. 3.14. Пакет Client содержит два пакета – пакет ClientGUI, в котором находится описание пользовательского интерфейса и пакет 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, в общем случае – что один пакет использует сущности, определенные в другом.

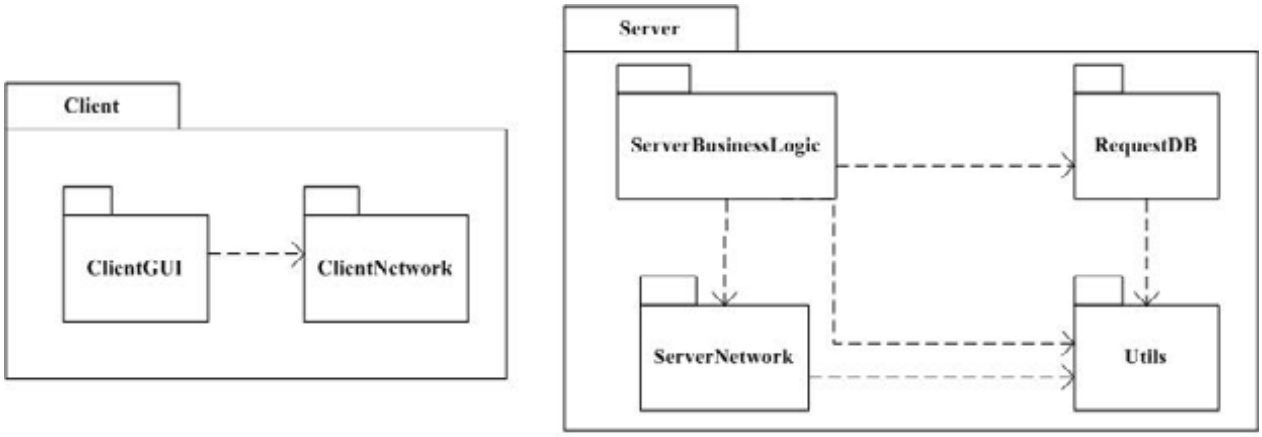



Рис. 3.14.

Пакет Server содержит все проекты нашего 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. Util является служебным пакетом (проектом в Visual Studio), где находятся все вспомогательные типы данных, классы, операции и т.д., которые используются всеми пакетами сервера. На рис. 3.15 показано, как раскрывается дальше пакет ServerBusinessLogic.

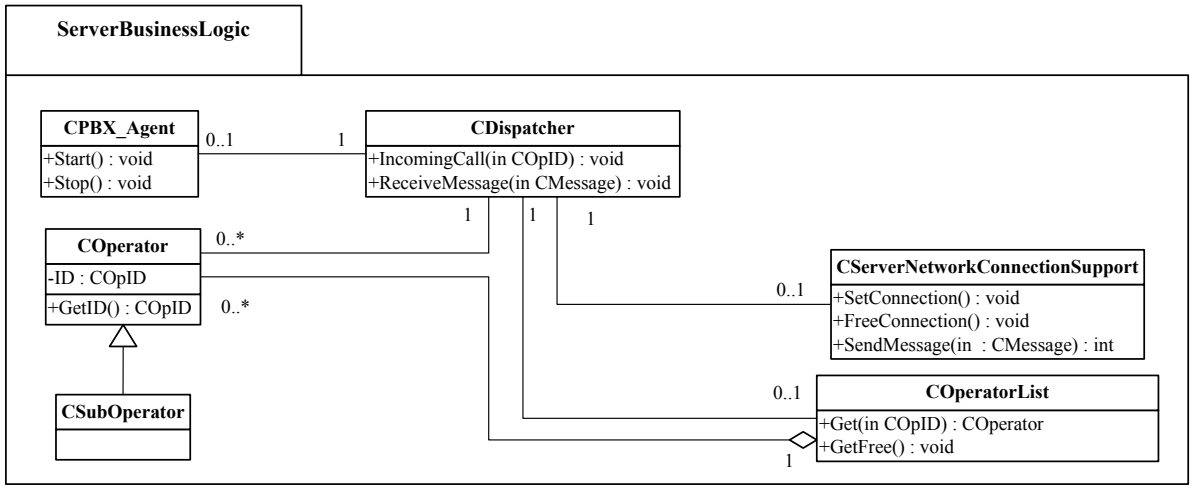



Рис. 3.15

В нашем случае пакеты почти полностью повторяют компоненты. Так произошло, потому что мы представляем упрощенную модель ПО “Телефонной службы приема заявок”. В действительности это приложение содержит около 50-ти различных проектов. Если бы мы создали полную модель пакетов, то она была бы не так похожа на модель компонент. Если компоненты удобно описывать в отдельном пакете, то не каждый пакет образует компоненту. Однако пересечения все равно бы были, ведь мы по-разному изображаем одно и то же.

Необходимо отметить, что модель пакетов с большой глубиной иерархии целесообразна для больших приложений, при проектировании таковых. Полезно как-то предварительно прикинуть структуру проектов нашего приложения, хотя, конечно, потом это видение может меняться. Но все равно полезно, чтобы модель пакетов находилась перед глазами разработчиков. Ее удобно также использовать при ознакомлении с исходным кодом системы.

**Диаграммы объектов (object diagrams).** Этот вид диаграмм предназначен для описания какого-либо фрагмента системы. На нем изображаются экземпляры классов – то есть те объекты, которые реально существуют в системе в некоторой момент ее работы. Пример такой диаграммы представлен на рис. 3.16.

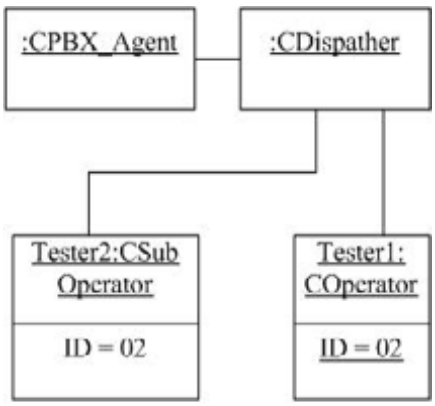



Рис. 3.16.

Понятно, что информация об экземплярах классов необходима вовсе не для спецификации системы (для этого используются, например, диаграммы классов), а для обсуждения некоторого ее фрагмента. Пусть есть, например, ошибка, возникшая на стыке разных подсистем, и каждый из разработчик подсистемы из этого множества «валит»

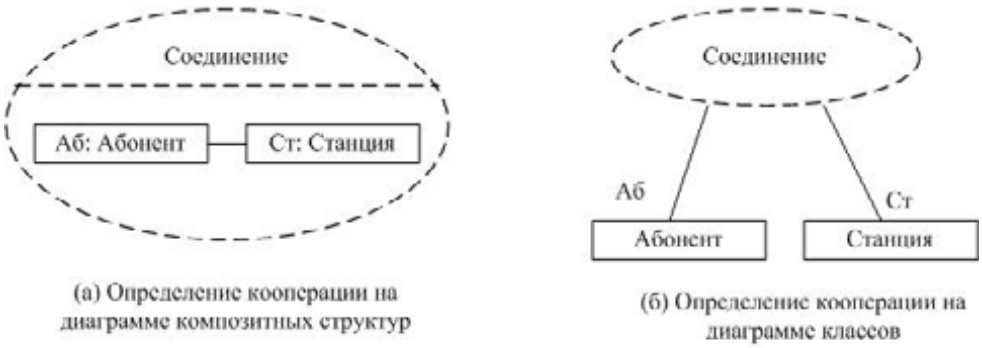
вину на других. Тестеровщик, нашедший ошибку, не может понять, кому поручить ее исправление. Тогда он собирает совещание, приглашает туда всех авторов «окрестных» подсистем, призывает, при необходимости и для поддержки менеджера проекта и пытается разрешить старый как мир вопрос: «Кто виноват?». При этом он может нарисовать диаграмму объектов для той части системы и для того момента ее существования, где проявляется данная ошибка. Эта диаграмма может помочь сконцентрировать внимание всех участников встречи на обсуждаемом вопросе.

**Диаграммы композитных структур (composite structure diagrams): кооперации.** Эти диаграммы предназначены для декомпозиции некоторых видов структурных элементов UML. То есть с их помощью можно строить иерархию описаний системы, постепенно, на каждом уровне, раскрывая детали. Из диаграмм, которые мы рассмотрели, структурную декомпозицию поддерживают лишь диаграммы пакетов. Но пакеты являются, скорее, внутримодельным средством создания иерархии описаний. А при создании больших систем нужны средства блочной декомпозиции самих систем, а не их моделей. Именно для этого и предназначаются диаграммы композитных структур, позволяя создавать иерархии кооперации, объектов, классов и компонент. Таким образом, фрагменты модели, отображаемые на диаграммах композитных структур, могут появляться на диаграммах объектов (кооперации, композитные объекты), классов (композитные классы, кооперации), на диаграммах компонент (композитные компоненты).

Вообще-то, в UML-стандарте сказано, с помощью этих диаграмм можно описывать произвольные конструкции UML, которые наследуются от конструкции метамодели Structured Classifier.

С этого момента, для дальнейшей демонстрации возможностей, мы перестаем рассматривать наш пример телефонной службы приема заявок. Крайне редко бывает так, что при проектировании или описании одной системы используются все виды диаграмм UML. Например, диаграммы классов удобны, если мы реализуем типичное объектно-ориентированное приложение. И при этом редко используются диаграммы состояний и переходов. В то же время последние очень активно применяются при разработке ПО для телекоммуникационных систем, совместно с диаграммами компонент и диаграммами композитных структур. При этом диаграммы классов могут не использоваться вовсе. И так далее. Поэтому, чтобы не запутывать читателей, мы, исчерпав данный пример с точки зрения возможностей UML, будем приводить другие примеры, подбирая их наиболее подходящим способом для оставшегося набора диаграмм.

Рассмотрим сначала кооперации. *Кооперация* (cooperation) – это описание определенной задачи (например, какой-либо пользовательской функции системы, или внутренней задачи ПО, или же какого-либо алгоритма предметной области) в терминах задействованных в ней элементов системы. Например, на рис. 3.17 показано определение кооперации «Соединение» между абонентом и телефонной станцией (в двух вариантах).




(а) Определение кооперации на диаграмме композитных структур

(б) Определение кооперации на диаграмме классов

Рис. 3.17.

То есть в нашем ПО есть два класса – Абонент и Станция. В числе той функциональности, которую они реализуют, есть та, которая отвечает за установку телефонного соединения абонента и станции. Именно этот аспект их функциональности и изображен на рис. 3.17. Та функциональность этих классов, которая входит в кооперацию «Соединение», обозначена ролями этих классов – «Аб» и «Ст» (см. раздел, где мы рассматривали диаграммы классов). На рис 3.17 (а) эти роли представлены прямоугольниками, на рис. 3.17 (б) – пометками на линиях, которые соединяют классы «Абонент» и «Станция» с кооперацией «Соединение».

В дальнейшем, мы можем использовать кооперацию «Соединение» на других диаграммах композитных структур, как показано, на рис. 3.18. Использование кооперации – это новая конструкция UML, которая соотносится с описанием кооперации также, как экземпляр класса и соответствующий класс.


Рис. 3.18. Применение кооперации на диаграмм объектов.

На рис. 3.18 изображена диаграмма объектов, куда вставлено использование кооперации Соединение. Два объекта – Вызывающий и Моя станция – «прицеплены» к ролям кооперации Аб и Ст. Чтобы это можно было сделать нужно, чтобы прицепляемые к кооперации модельные сущности были бы подтипами ролей ассоциации.

Кооперация может быть использована при конструировании другой, более сложной кооперации, как показано на рис. 3.19.


Рис. 3.19.

На этом рисунке определяется кооперация «Соединение абонентов одной станции» и при этом 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».

Мы не будем тут останавливаться на композитных классах и объектах. Скажем только, что диаграммы композитных структур являются альтернативным средством изображения класса/объекта, который сильно агрегирует другие классы/объекты. Сильная агрегация, другими словами, композиция, рассматривалась нами при обсуждении диаграмм классов.

**Диаграммы композитных структур (composite structure diagrams): компоненты.** Познакомимся теперь с композитными компонентами, которые широко используются при моделировании «многослойных» телекоммуникационных систем. На рис 3.20 (а) показана упрощенная архитектура автоматизированного телевещательного комплекса, который управляет различным вещательным оборудованием (в частности, видеомагнитофонами), автоматизируя монтаж материала и его вещание в эфир в реальном времени, с разных видеоисточников. Система делится на пять уровней. Первый уровень – это оборудование (компонента «Оборудование»), второй – ПО, которое непосредственно работает с оборудованием (компонента «Драйвера»), третий уровень – это компонента «Плэй», поддерживающая программную работу с высокоуровневыми функциями источников видео, например, реализуя такие команды, как перемотать кассету, начать запись, перемотать до такого-то кадра и т.д. Четвертый уровень (компонента «Сервисы») отвечает за целевые сервисы системы, например, перезапись материала с одно источника на другой, вещание плэй-листа в эфир, когда отдельные его материалы находятся на разных видеоисточниках и т.д. Последний, пятый уровень (компонента «Клиентские приложения») соответствует за прикладному, пользовательскому ПО, которое использует целевые сервисы системы и для работы с аппаратурой комплекса, предоставляет пользователю экранный интерфейс и пр.

На этом рисунке видно, что компоненты общаются друг другом через интерфейсы, как мы обсуждали, рассматривая диаграммы компонент. Однако тут у нас появляется дополнительные конструкции под названием порты, из которых, как это видно на рис. 3.20 (а) и (б), «растут» интерфейсы (как конструкция предоставления интерфейса, так и

конструкция использования интерфейса). Порты нам понадобились потому, что все наши компоненты внутри также состоят из отдельных компонент. На рис. 3.20 (б) показана внутренняя структура для компонент «Оборудование» и «Драйвера». Порты нужны, чтобы можно было провязывать части компоненты с его границей, и уже от границы предоставлять внешние интерфейсы. Ведь внутренность компоненты наружу не видна, все ее взаимодействие с внешним миром происходит через имеющиеся на границе компоненты порты.

Порт – это точка, через которую происходит взаимодействие. К этой точке подходят соединения с наружи компоненты и изнутри, с портом связан один или несколько интерфейсов, порт может иметь множественность (то есть осуществлять не одно, а несколько соединений). Теперь подробно рассмотрим рис. 3.20 (б).

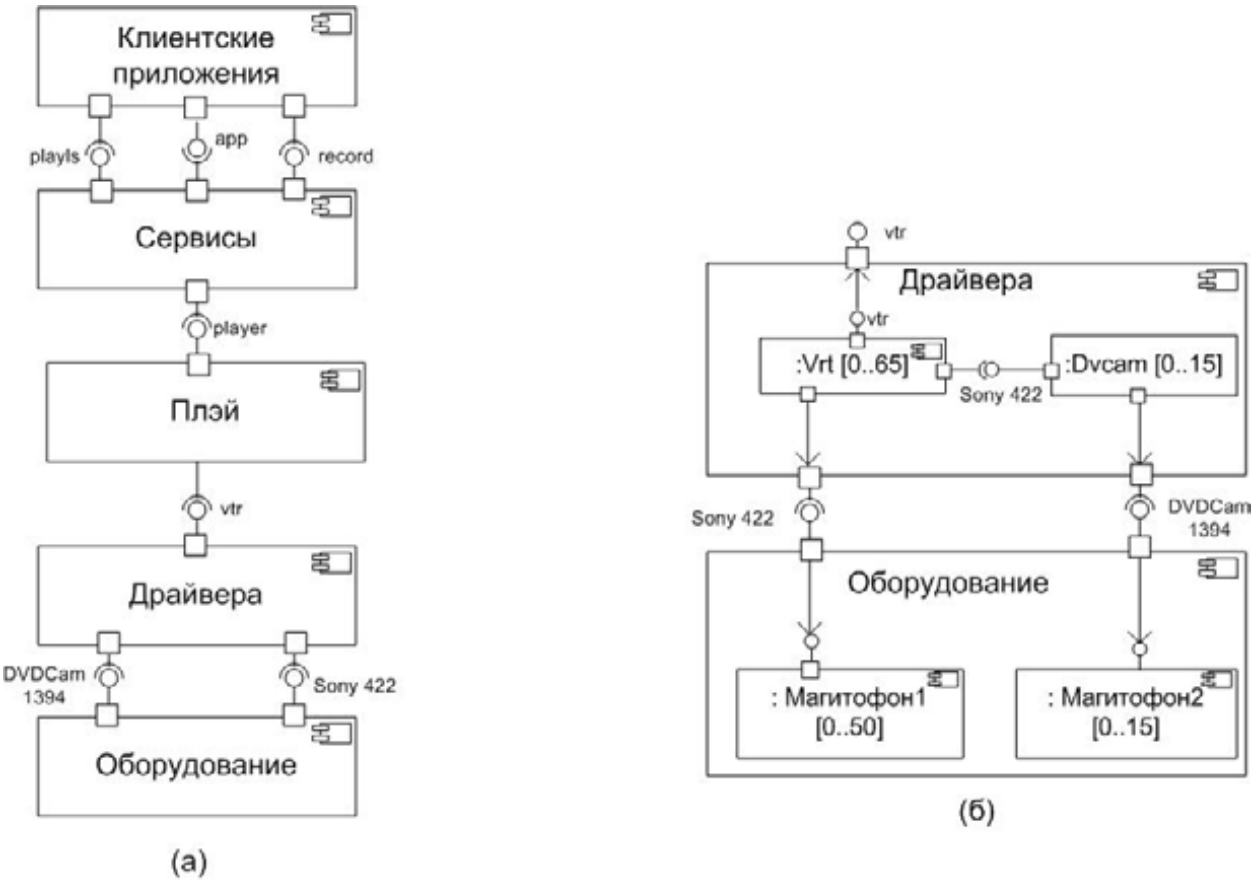



Рис. 3.20.

В компоненте «Оборудование» есть две другие компоненты, которые имеют тип «Магнитофон 1» и «Магнитофон 2». Эти магнитофоны отличаются теми интерфейсами, которые у них есть для программного управления. В первом случае это Sony422, во втором случае – Dvdcam1394. Указано, сколько штук магнитофонов каждого типа может быть в системе. Ограничения определяются типовыми размерами стойки оборудования. Компонента «Драйвер» содержит компоненту-переходник Dvdcam, которая является преобразователем сигналов из интерфейса Sony 422 в сигналы интерфейса DVDCam1394. Поэтому таких компонент должно быть столько же, сколько и компонент «Магнитофон2» - по одному преобразователю на каждый магнитофон этого типа. Компоненты типа Vtr осуществляют нижнеуровневое управление магнитофонами по интерфейсу Sony 422 – как магнитофонов типа «Магниофон1», так и магнитофонов типа «Магниофон2» (последних – через компоненту-переходник Dvdcam). Каждому магнитофону

соответствует одна такая компонента. Следующим выше компонентам предоставляется более высокоуровневый интерфейс управления магнитофонами через компоненту vtr.

Отметим, что композитным является тип компоненты. А он, в свою очередь, описывается через компоненты-роли, аналогично тому, как кооперация описывалась ролями классов. У ролей-компонент могут быть порты и интерфейсы, определенные у их типов-компонент, а также может указываться количество возможных экземпляром и некоторые другие свойства. Таким образом, на одной диаграмме можно изобразить два уровня вложенности, но не более. Дальше нужно создавать отдельную диаграмму для типов ролей компонент, которые бы использовались на втором уровне (конечно, можно эти типы компонент определять на этой же диаграмме, но – отдельно, рядом).

**Диаграммы конечных автоматов (statechart diagrams).** На рис. 3.21 приведен пример диаграммы конечных автоматов. Эта диаграмма описывает алгоритм поведения объектов класса COperator системы «Телефонной службы приема заявок», изображенного на рис. 3.15.

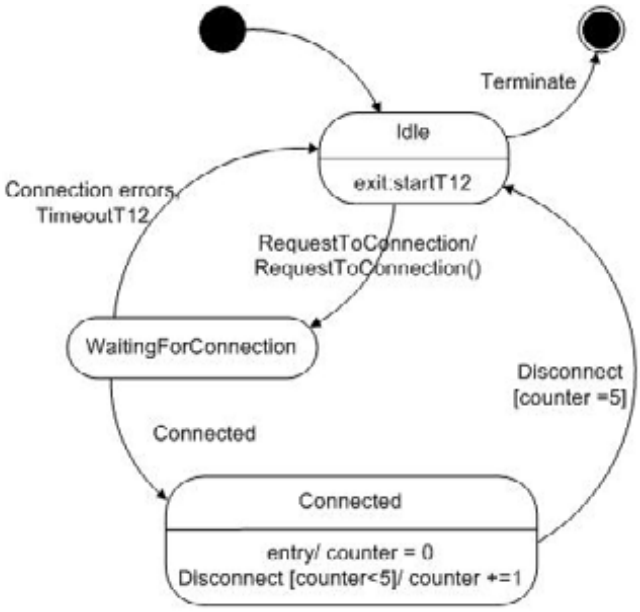



Рис. 3.21.

После инициализации объекта он переходит в состояние Idle. В этом состоянии объект пребывает, пока свободен, то есть 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, то есть указание завершить всю свою работу и освободить оперативную память.

**Временные диаграммы (timing diagrams).** Этот вид диаграмм предназначен для наглядного изображения потока изменеия состояний нескольких объектов. Такая возможность полезна при моделировании встроенных систем. Рассмотрим пример.

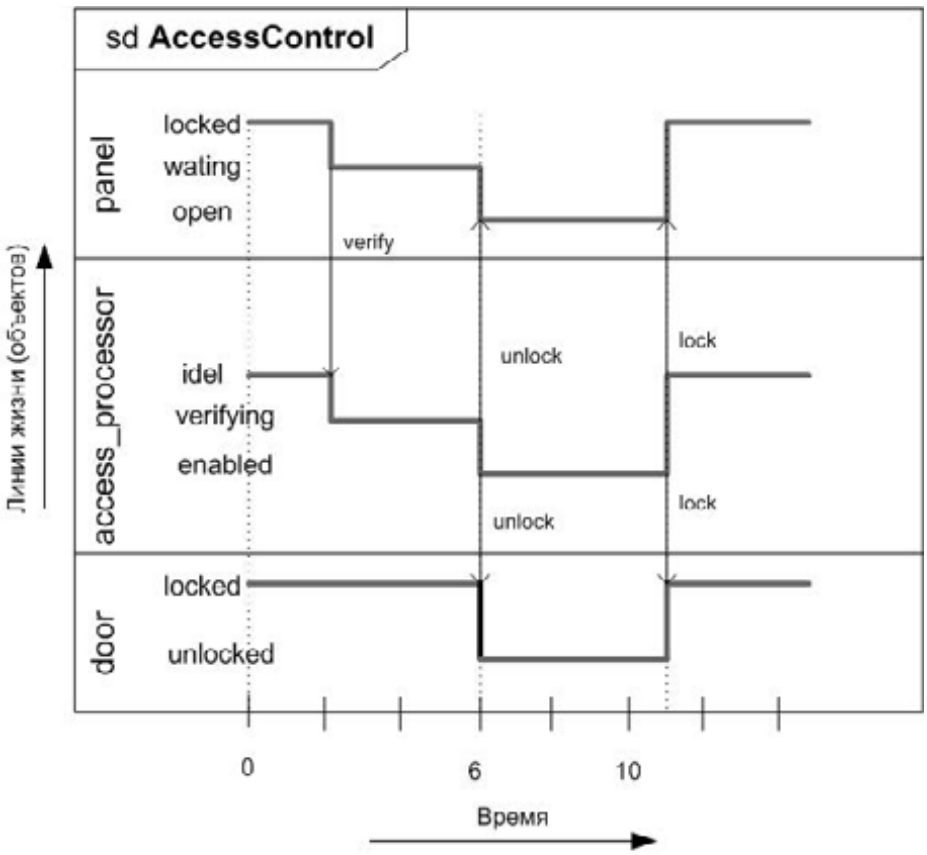



Рис. 3.22.

На рис. 3.22 показан фрагмент работы системы AccessControl, которая управляет открытием/блокированием двери в помещение по предъявлению человеком электронного ключа. У этой системы изображено три компоненты. Первая, panel, является устройством, у которого есть дисплей для отображения текущего состояния все 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, после получения этого сообщения он переходит в состояние verify. 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 (то есть переходит в состояние unlock). Открытой она остается ровно 5 секунд, после чего процессор присылает ей команду lock и она закрывается, то есть снова переходит в состояние locked. Одновременно процессор посылает команду lock также и компоненте panel, которая переходит в свое исходной состояние locked.

Видно, что на временных диаграммах, также как на диаграммах последовательностей и коммуникаций, показываются только главные ветки алгоритмов, то есть отсутствуют ветвления. Объекты и их состояния откладываются по оси ординат,

время – по оси абсцисс. Время градуировано в какой-либо шкале измерений. В нашем примере каждое деление соответствует двум секундам.

Область каждого объекта – это прямоугольник отделенный от другого, соседнего (другого объекта) прямой линией, параллельной оси абсцисс. Объекты могут обмениваться сообщениями, через которые происходит синхронизация их поведения. Сообщения изображаются вертикальными линиями со стрелками (вверх или вниз).

**Диаграммы схем взаимодействия (interaction overview diagram).** Этоn тип диаграмм является смесью диаграмм активностей и диаграмм последовательности. Вместо действий в узлы диаграмм активностей подставляются диаграммы последовательности (сценарии). Таким образом, достигается цель задавать сложное поведение с ветвлениями, так как иначе на диаграммах последовательности ветвления не задать.

## Основная литература


1. Г.Буч, А.Якобсон, Дж. Рамбо. UML. – Изд. 2-ое. Питер, 2006. – 735 с.
2. UML 2.0 Infrastructure Specification, September, 2004. http://www.omg.org/
3. Д.В. Кознов. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. – Изд-во СПбГУ, 2004. – 143 с.