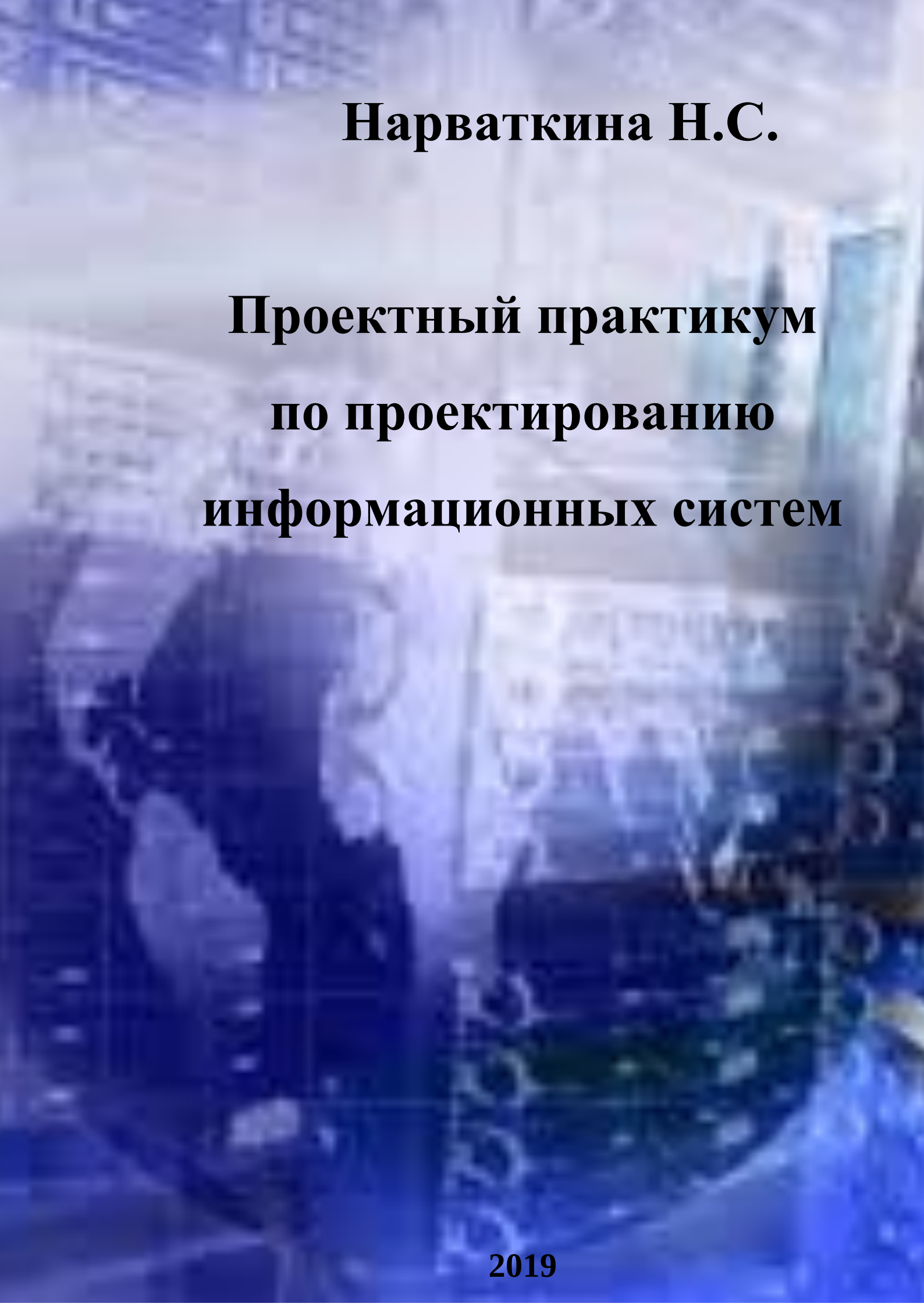
Нарваткина Н.С.

Проектный практикум по проектированию информационных систем



2019

Содержание

Введение

В настоящее время унифицированный *язык моделирования* - *UML*, является, пожалуй, самой модной технологией в области программной инженерии. Почему это так? Дело в том, что *UML* позволяет системным архитекторам представлять свое видение системы в виде набора стандартных диаграмм, которые, к тому же, служат отличным средством коммуникации в команде разработчиков и прекрасным помощником в общении с заказчиком. И при всем этом, *UML* - достаточно логичная и простая для изучения *нотация*, навыками использования которой, без сомнения, должен овладеть любой специалист, собирающийся работать в области программной инженерии. *Знание UML* нужно разработчикам, системным архитекторам, менеджерам…

С другой стороны, зачастую у нас просто нет времени на чтение руководств и подробное изучение документации, нет времени на вопросы, а нужно быстро получить ответы на них. Нужно быстро составить *представление* о технологии, познакомиться с ней на концептуальном, понятийном уровне, составить *представление* о *UML*, убедиться в его простоте и полезности, поверить в свои способности к моделированию, наметить направления дальнейшего совершенствования навыков и знаний, понять основополагающие концепции рассматриваемой технологии, на которые в дальнейшем Вы сможете "нарастить" более конкретные знания.

Пособие, которое вы держите в руках, базируется на двух очень простых принципах:

1. Каждое занятие имеет четкую структуру - сначала мы говорим, о чем пойдет речь, затем приводим список вопросов, которые будем обсуждать, а в конце каждой из них вас ожидают краткие итоги, список использованных источников, а также контрольные вопросы и упражнения.

2. Еще одно отличие этого пособия - его модульность. Каждое занятие полностью самодостаточно, и его можно читать отдельно, в отрыве от предыдущих и последующих. Таким образом, Вы получаете возможность изучать материал в любой последовательности.

3. Речь в пособии идет только о тех элементах *UML*, знать которые абсолютно необходимо. *По* структуре оно соответствует карте покрытия тем стандарта *UML* экзаменом UM0-100, которую всегда можно найти на сайте *OMG*: http://www.omg.org/uml-certification/UML_2-ToC-Fundamental.pdf.

На первом занятии познакомимся с краткой историей *UML*, его назначением, способами использования языка, структурой его определения, терминологией и нотацией.

На втором узнаем о том, какие виды диаграмм существуют в *UML*, какие из них наиболее часто используются и для чего предназначена каждая из них. Также здесь будут даны некоторые рекомендации относительно последовательности построения диаграмм.

Последующие четыре занятия посвящены более подробному рассмотрению диаграмм, каждому из наиболее часто используемых видов.

Итак, третье занятие расскажет о технологиях реализации основных принципов объектно-ориентированного подхода, подробнее расскажет о диаграмме классов - конечном результате проектирования и отправной точке процесса разработки.

На четвертом занятии речь пойдет о диаграммах активностей, которое напомнит многим блок-схемы алгоритмов. Но это не совсем, а вернее, совсем не блок-схемы!

Пятое занятие познакомит с диаграммами взаимодействия, позволяющими описать поведенческие аспекты системы, научит читать и строить диаграммы последовательностей и кооперации.

На шестом занятии поговорим о концептуальном проектировании, о том, как описать систему с точки зрения пользователя, - о диаграммах прецедентов.

Хотя для выполнения упражнений Вам потребуется лишь *лист* бумаги и карандаш, вы должны знать, что существует огромное количество программного обеспечения, так называемых CASE-систем для построения диаграмм *UML*. О наиболее популярных из них будет рассказано на седьмом занятии. Причем мы постараемся рассмотреть и признанных лидеров рынка, и его "аутсайдеров", и коммерческих "монстров", и "легкие" программы с открытым исходным кодом.

В заключении подводятся некоторые итоги, указываются направления для дальнейшего совершенствования своих знаний в области ООАП и *UML*.

Занятие 1. UML. Основные понятия

Назначение языка

UML - унифицированный *язык моделирования*. Из этих трех слов главным является *слово* " **язык** ". Что же такое язык? Не будем изобретать велосипед, а лучше заглянем в *глоссарий*, благо в Интернете их величайшее множество. Сделав это, мы скорее всего обнаружим *определение*, подобное приведенному ниже.

Язык - система знаков, служащая:

- средством человеческого общения и мыслительной деятельности;
- способом выражения самосознания личности;
- средством хранения и передачи информации.

Язык включает в себя набор знаков (словарь) и правила их употребления и интерпретации (грамматику).

К этому достаточно исчерпывающему определению нужно добавить, что языки бывают естественные и искусственные, формальные и неформальные. *UML* - язык формальный и искусственный, хотя, как мы увидим далее, этот ярлык к нему не совсем подходит. Искусственный он потому, что у него имеются авторы, о которых мы еще не раз упомянем в дальнейшем (в то же время, развитие *UML* непрерывно продолжается, что ставит его в один ряд с естественными языками). Формальным его можно назвать, поскольку имеются правила его употребления (правда, описание *UML* содержит и явно неформальные элементы, как мы, опять-таки, позже увидим). Еще один нюанс: *UML* - язык графический, что также немного путает ситуацию!

При описании формального искусственного языка, что мы уже видели на примерах описания языков программирования, как правило, описываются такие его элементы, как:

- синтаксис, то есть определение правил построения конструкций языка;
- семантика, то есть определение правил, в соответствии с которыми конструкции языка приобретают смысловое значение;
- прагматика, то есть определение правил использования конструкций языка для достижения нужных нам целей.

Естественно, *UML* включает все эти элементы, хотя, как мы опять-таки увидим далее, в их описании тоже наблюдаются отличия от правил, принятых в языках программирования.

Второе *слово* в фразе, которой расшифровывается аббревиатура *UML* - *слово* " **моделирование** ". Да, *UML* - это *язык моделирования*. Причем *объектно-ориентированного моделирования*. Более подробно о смысле понятия "*моделирование*" мы поговорим чуть позже, а пока отметим, что *слово* это весьма многозначно. В английском языке есть целых два слова

- **modeling** и **simulation**, которые оба переводятся как "*моделирование*", хотя означают разные понятия. *Modeling* подразумевает создание модели, лишь описывающей *объект*, а *simulation* предполагает получение с помощью созданной модели некоторой дополнительной информации об объекте. *UML* в первую *очередь - язык моделирования* именно в первом смысле, то есть средство построения *описательных моделей*. Как средство симулирования его тоже можно использовать, хотя для этой роли он подходит не так хорошо.

Третье *слово* в названии *UML - слово* " **унифицированный** ". Его можно понимать тоже неоднозначно. В литературе можно встретить описание эры "до *UML*" как "войны методов" моделирования, ни один из которых "не дотягивал" до уровня индустриального стандарта. *UML* как раз и стал таким единым универсальным стандартом для *объектно-ориентированного моделирования*, которое во времена его создания как раз "вошло в моду". "Единым" языком моделирования *UML* можно назвать еще и потому, что в его создании, как мы увидим далее, объединились усилия авторов трех наиболее популярных методов моделирования (и не только их).

Подводя итоги, кратко можно сказать, что *UML* - искусственный язык, который имеет некоторые черты естественного языка, и *формальный язык*, который имеет черты неформального. Это звучит не очень понятно, но это действительно так!

Историческая справка

Откуда взялся The *UML*? Если говорить коротко, то *UML* вобрал в себя черты нотаций Грейди Буча (Grady Booch), Джима Румбаха (Jim Rumbaugh), Айвара Якобсона (Ivar Jacobson) и многих других.

В не такие уж и далекие 80-е годы было множество различных методологий моделирования. Каждая из них имела свои достоинства и недостатки, а также свою нотацию. То смутное время получило название "войны методов". Проблема в том, что разные люди использовали разные нотации, и для того чтобы понять, что описывает та или иная *диаграмма*, зачастую требовался "переводчик". Один и тот же символ мог означать в разных нотациях абсолютно разные вещи! На рисунке ниже можно увидеть лишь малую часть многообразия методов, которые существовали в то время и в какой-то мере повлияли на *UML (*рисунок 1.1).

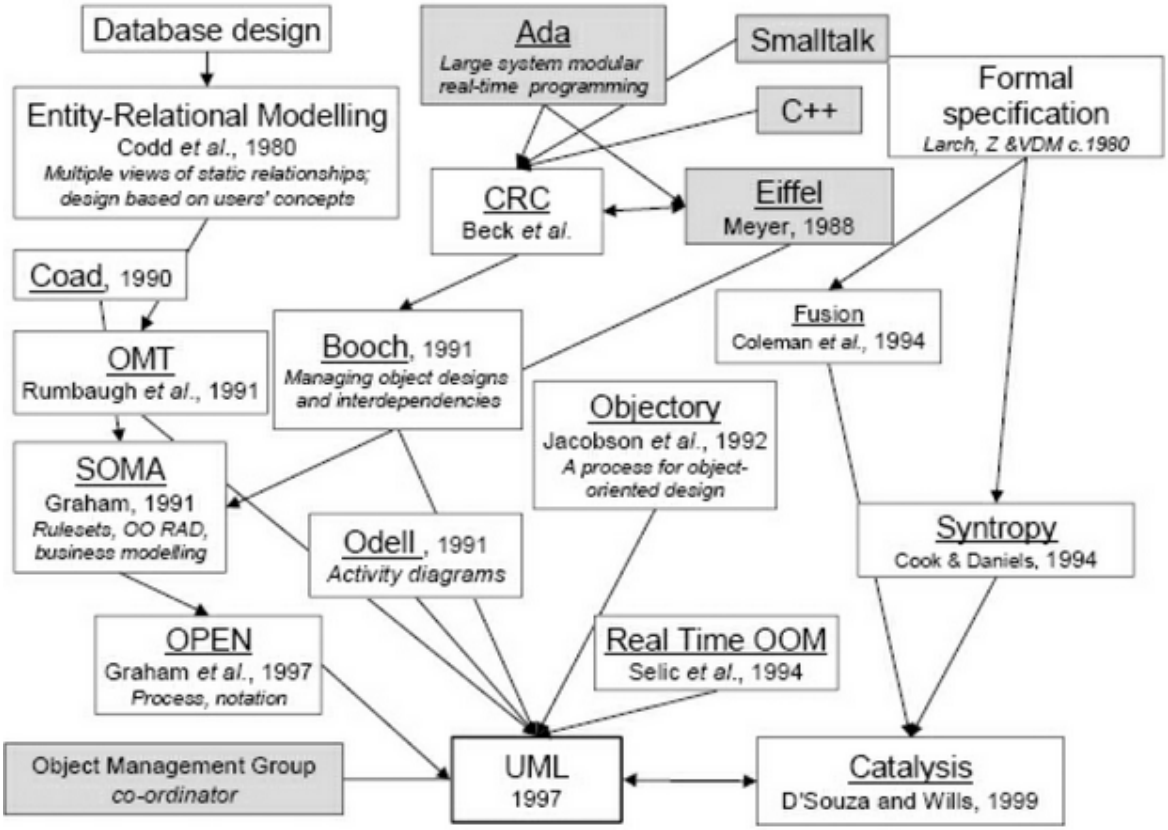


Рисунок 1.1.

К тому же примерно в это же время (начало 80-х) стартовала "объектно-ориентированная эра". Все началось с появлением семейства языков программирования SmallTalk, которые применяли некоторые понятия языка Simula-67, использовавшегося в 60-х годах. Появление объектно-ориентированного подхода в первую *очередь* было обусловлено увеличением сложности задач. *Объектно-ориентированный подход* внес достаточно радикальные изменения в сами принципы создания и функционирования программ, но, в то же время, позволил существенно повысить *производительность* труда программистов, *по*-иному взглянуть на проблемы и методы их решения, сделать программы более компактными и легко расширяемыми. Как результат, языки, первоначально ориентированные на традиционный подход к программированию, получили ряд объектноориентированных расширений. Одной из первых, в середине 80-х, была *фирма* Apple со своим проектом *Object Pascal*. Кроме этого, *объектно-ориентированный подход* породил мощную волну и абсолютно новых программных технологий, вершинами которой стали такие общепризнанные сегодня платформы, как Microsoft *.NET* Framework и *Sun Java*.

Но самое главное, что появление *ООП* требовало удобного инструмента для моделирования, единой нотации для описания сложных программных систем. И вот "три амиго", три крупнейших специалиста, три автора наиболее популярных методов решили объединить

свои разработки. В 1991-м каждый из "трех амиго" начал с написания книги, в которой изложил свой метод ООАП. Каждая методология была *по*-своему хороша, но каждая имела и недостатки. Так, метод Буча был хорош в проектировании, но слабоват в анализе. *OMT*Румбаха был, наоборот, отличным средством анализа, но плох в проектировании. И наконец, Objectory Якобсона был действительно хорош с точки зрения *user experience*, на который ни метод Буча, ни *OMT* не обращали особого внимания. Основной идеей Objectory было то, что *анализ* должен начинаться с прецедентов, а не с *диаграммы классов*, которые должны быть производными от них.

К 1994-му существовало 72 метода, или частные методики. Многие из них "перекрывались", т. е. использовали похожие идеи, нотации и т. д. Как уже говорилось выше, чувствовалась острая потребность, "социальный заказ" - закончить "войну методов" и объединить в одном унифицированном средстве все лучшее, что было создано в области моделирования.

А что сейчас? The *UML* живет и развивается. Сейчас мы имеем *UML* 2.0 и десятки CASE-средств, поддерживающих *UML*, о многих из которых будет рассказано в "Обзор CASE-средств для построения диаграмм UML" . Вопреки популярному мнению, в наши дни Rational не владеет *UML*, но продолжает работать над ним. *UML* же принадлежит *OMG*, а сама Rational ныне является одним из подразделений *IBM* и фигурирует во всех документах как *IBM* Rational. *UML* же получил множество пакетов расширений, называемых *профайлами* и позволяющих использовать его для моделирования систем из специфических предметных областей.

Способы использования языка

И вот Румбах присоединился к Бучу в Rational Inc. Они объединили свои нотации и создали первую версию *UML*. В 1995 году на конференции OOPSLA они представили его как *UnifiedMethod*, который потом и получил название *UML*. Чуть позже к ним присоединился Якобсон, который добавил к результатам их труда элементы Objectory и начал работу над Rational *Unified* Process (*RUP*). В 1997 году *UML* был отправлен в *Object Management Group* (*OMG*) для стандартизации. Кроме трех нотаций "трех амиго" *UML* вобрал в себя элементы многих других методологий, что опять-таки хорошо видно из рисунка, приведенного выше.

Начать хотелось бы с демонстрации известной картинки, которая уже более двух десятилетий "живет" в Интернете, но источник ее никому не известен (если кто-то из читателей сможет пролить свет на ее происхождение, *автор* будет очень благодарен за информацию). Эта картинка прекрасно иллюстрирует типичный процесс создания продукта, или "решения" (поскольку продукт решает проблему заказчика), как любят говорить в Microsoft (рисунок 1.2).

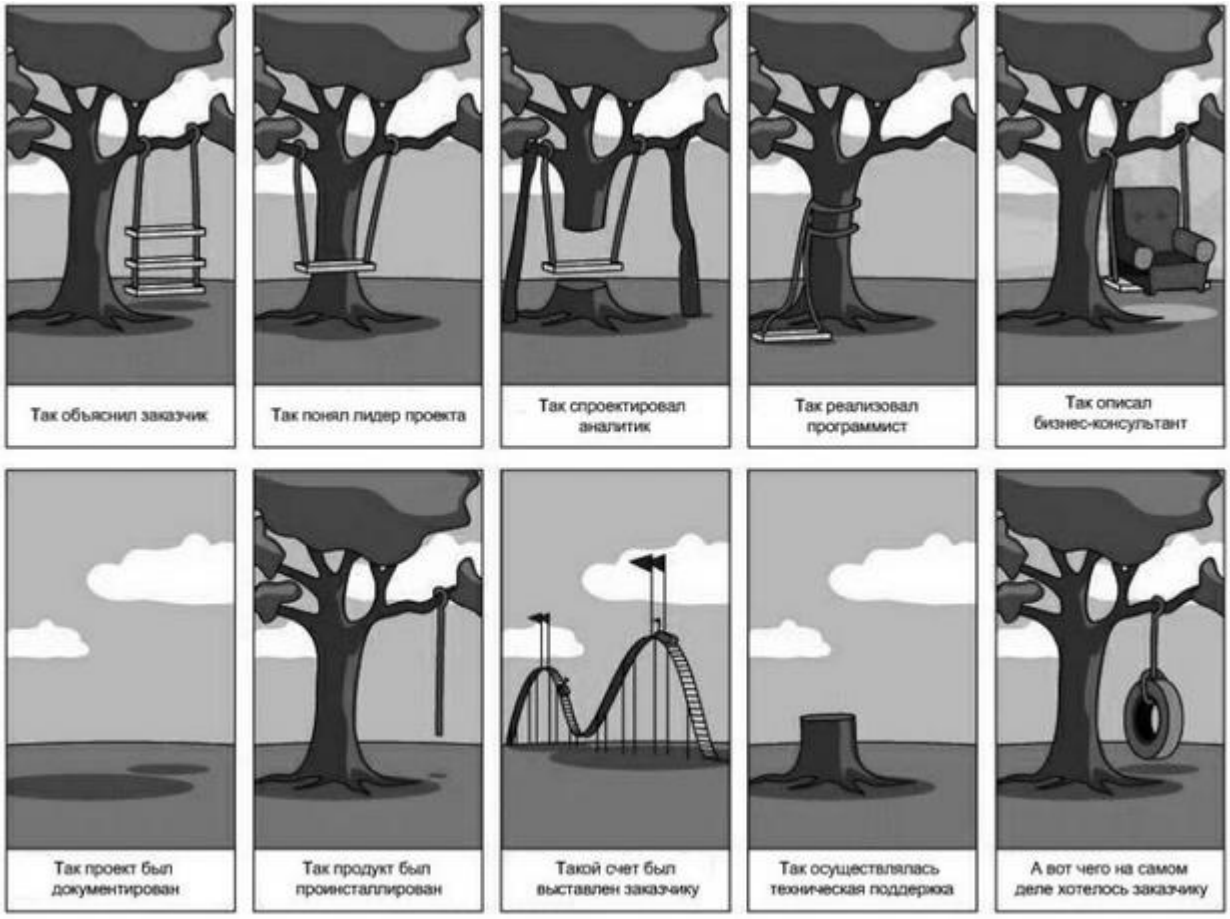


Рисунок 1.2.

Здесь мы видим все проблемы программной инженерии, в частности проблемы с коммуникацией и пониманием, вызванные отсутствием четкой спецификации создаваемого продукта. Так вот, авторы *UML* определяют его как графический *язык моделирования* общего назначения (т. е. его можно применять для проектирования чего угодно - от простой качели, как на рисунке, до сложного аппаратно-программного комплекса или даже космического корабля), предназначенный для **спецификации**, **визуализации**, **проектирования** и **документирования** всех артефактов, создаваемых в ходе разработки.

Итак, *UML* в первую *очередь* - это спецификации. Заглянем снова в *глоссарий*.

Спецификация - подробное описание системы, которое полностью определяет ее цель и функциональные возможности.

Различают:

- словесные спецификации на естественном языке;
- модельные спецификации;
- формальные спецификации.

Не следует также забывать, что заказчик и разработчик имеют, как правило, абсолютно разное понимание смысла этого артефакта. А ведь кроме этого есть еще аналитики,

менеджеры, бизнес-консультанты... Каждый из них называет спецификации *по*-своему: *постановка задачи*, требования пользователя, *техническое задание*, *функциональная спецификация*, *архитектура* системы... Причем все эти люди, являясь специалистами в абсолютно разных предметных областях, говорят каждый на своем языке и зачастую просто не понимают друг друга. Вот потому-то и возникает проблема, представленная на рисунке, проблема, которую может решить только наличие единого, унифицированного средства создания спецификаций, достаточно простого и понятного для всех заинтересованных лиц.

Как уже говорилось выше, различают спецификации трех видов. *Словесные спецификации на естественном языке* как раз и вызывают массу проблем, поскольку создаются разными специалистами на "их языке". Другим видом спецификаций являются *формальные* спецификации. Действительно, описание спецификации с помощью строгого математического языка было бы чудесным решением всех проблем, т. к. сам способ записи исключал бы малейшие неоднозначности. Да, в математике есть, например, *алгебра* высказываний, с помощью которой можно пытаться создавать технические задания на разработку некоторых приложений. Проблема кроется в слове "некоторых". Понятно, что *формальная спецификация* является, *по* сути, математической *моделью задачи* и потому для вычислительных задач все выглядит достаточно просто. Формализация же задач из других областей знаний может оказаться более сложной и трудоемкой проблемой, чем разработка самого приложения ввиду отсутствия четкой математической модели. Один из принципов прикладной "мерфологии" гласит, что лучшей *спецификацией программы* является ее текст. Так же, как и большинство остальных законов Мерфи, это утверждение просто поражает своей правдивостью...

Когда мы говорим о том, что *UML* - это средство *визуализации*, мы имеем в виду *модельные* спецификации. Все мы знаем, как иногда трудно заставить себя "вникнуть" в суть материала, излагаемого в очередном учебнике или мануале. Изучение чего-то нового идет гораздо проще, если документ содержит не только текст, а еще и иллюстрации к нему. А если руководство или учебник выглядят как картинки с подписями (вспомните майкрософтовские учебники и трейнер-киты или руководства пользователя мобильных телефонов!), то усвоение нового материала происходит еще проще и эффективнее. Недаром до сих пор так популярны комиксы, которые также представляют собой картинки с текстом!

Так вот, такие картинки с подписями наглядны и интуитивно понятны, причем почти однозначно понимаются любыми заинтересованными лицами, так что могут использоваться в качестве средства общения между людьми. *UML* позволяет создавать такие простые и понятные картинки (модели), описывающие систему с разных сторон, которые можно показать заказчику и обсудить с ним, т. е. служит средством коммуникации в команде. Посмотрите на рисунок ниже (рисунок 1.3). Все ведь понятно, правда?

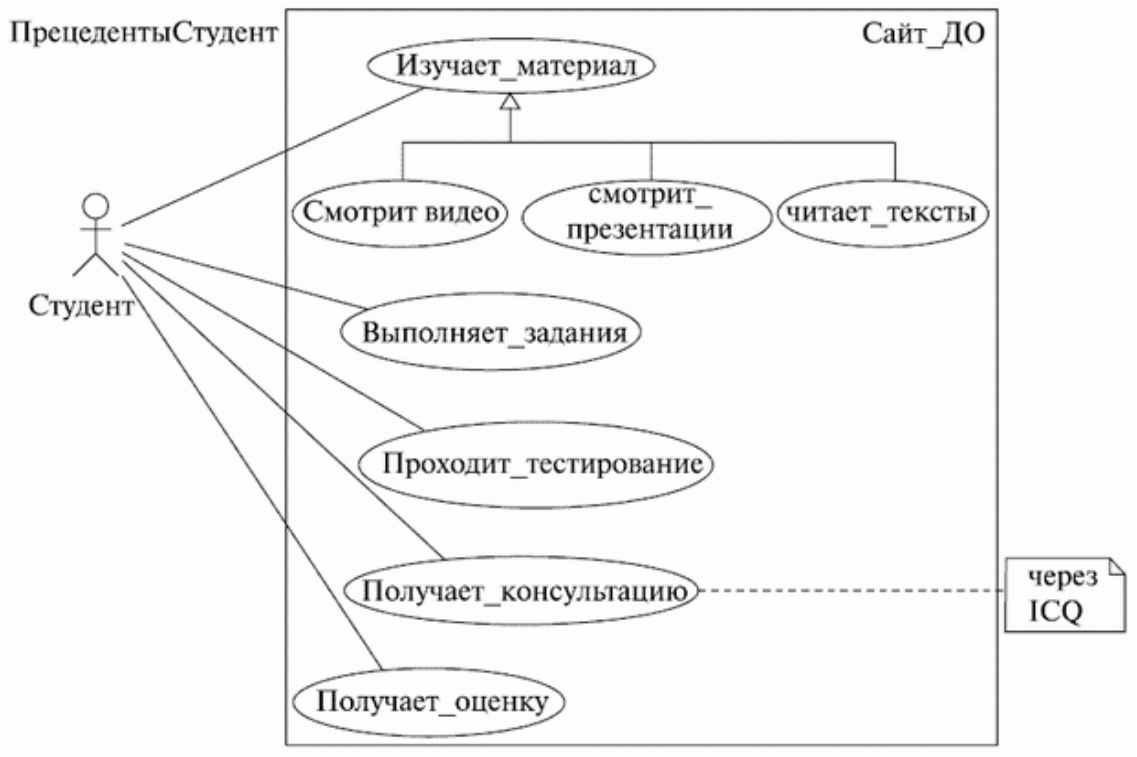


Рисунок 1.3.

Перейдем к *проектированию*. Да, *UML* позволяет строить модели программных систем (вообще говоря - ЛЮБЫХ систем). *По* этим моделям потом может производиться генерация каркасного кода проектируемых приложений. Более того, возможен процесс, который часто называют "реверс-инжинирингом", - т. е. создание *UML*-модели из существующего кода приложения. Не будем сейчас обсуждать качество получающегося кода или моделей при реверс-инжиниринге. Пока оно весьма далеко от идеала, но ведь технологии и инструменты постоянно совершенствуются, так что можно надеяться, что когда-нибудь мы сможем создавать приложения визуально, не прибегая к языку программирования, а пользуясь лишь *UML*...

И последнее из этого набора слов - " *документирование* ". *По* большому счету, *UML*-модели сами *по* себе уже являются документами (и весьма понятными, даже для неспециалиста, как мы уже могли убедиться, посмотрев на предыдущий рисунок; кроме этого, как мы еще упомянем далее, модели *UML* являются XML-документами). Причем любой элемент на любой диаграмме может быть снабжен ноутсом - текстовым комментарием. Т. е. построение набора диаграмм уже является процессом документирования будущей системы. Более того, большинство инструментов *UML*-проектирования умеют извлекать текстовую информацию из моделей и генерировать относительно удобочитаемые тексты.

Итак, подводя итоги, скажем, что *UML можно* использовать для рисования картинок, которые можно использовать для коммуникаций внутри команды и в ходе взаимодействия с заказчиком, т. е. он может служить средством обмена информацией. Кроме этого, как мы уже

говорили, *UML* является отличным средством спецификации систем, причем спецификации в процессе разработки. Разработанные архитектурные решения, задокументированные с помощью *UML*, могут быть использованы повторно (что сейчас также очень "модно"). Как уже упоминалось выше, о таких вещах, как генерация кода, симуляция и *верификация моделей*, пока серьезно говорить не приходится, но в будущем, надеемся, будет и это...

Теперь о том, для чего *UML* использовать нельзя, вернее, чем он не является. Во-первых, *UML* не является языком программирования, хотя существуют средства выполнения *UML*-моделей как интерпретируемого кода (Unimod, FLORA и др.) и возможна, как уже говорилось выше, *кодогенерация*. Несмотря на это, *UML* - средство не программирования, а моделирования, т. е. создания не *программ*, а *моделей* любого уровня абстракции для систем из любой *предметной области*. Во-вторых, *UML* не является и спецификацией какого бы то ни было инструмента моделирования, хотя такие инструменты (и в больших количествах) имеются. Например, TAU G2 (с помощью которого создано большинство диаграмм в этом курсе), Borland Together, Poseidon, *Enterprise Architect, IBM* Rational Rose, *Dia,* Visio и др. Каким образом то или иное CASE-средство реализует *UML-моделирование*, никак не регламентируется и определяется самими разработчиками этих инструментов. И, наконец, в-третьих, *UML* не является и моделью какого-либо процесса разработки, даже Rational *Unified* Process (*RUP*), который был описан именно с помощью *UML* (а точнее, с помощью SPEM - профайла *UML*). *UML* можно использовать независимо от того, какую методологию разработки *ПО* вы используете, и даже если вы вообще не пользуетесь никакой методологией!

Структура определения языка

Здесь нам хотелось бы рассказать о том, как описан *UML* его авторами. Но прежде нужно поговорить о способах описания искусственных языков вообще (например, языков программирования).

Конечно, вы уже читали книги, в которых описывались языки программирования, и не могли не заметить, как авторы этих книг все время самоотверженно балансируют между точностью и понятностью описания. Велик соблазн описать язык формально точно, но такое описание своей сложностью может отпугнуть потенциального пользователя новой технологии. С другой стороны, "понятное", неформальное описание языка может получиться очень длинным и неполным и просто запутать читателя.

Как же определен *UML*? Довольно часто компиляторы и *IDE* языков программирования написаны с использованием этих же языков (вспомните хотя бы Turbo *Pascal*!). Подобный метод применяется и при описании *UML*. Авторы использовали так называемое четы-

рехуровневое мета-*моделирование*. Первый уровень - это сами данные. Второй - это их модель, т. е., например, описание их в программе. Третий - *метамодель*, т. е. описание языка построения модели. Четвертый - мета-*метамодель*, т. е. описание языка, на котором описана *метамодель*. Для примера - следующий рисунок, позаимствованный из стандарта *UML*, показывает применение этого подхода к простым записям о котировках акций (рисунок 1.4).

UML, как уже говорилось выше, описывается подобным образом. *Метамодель* - описание самого языка, мета-*метамодель* - описание формализма, с помощью которого производится описание языка. Все это сопровождается комментариями на естественном языке и примерами моделей. Организованное таким образом описание *UML* распространяется *OMG* абсолютно свободно и "лежит" на сайте *OMG, по* адресу http://www.omg.org/. Этот грандиозный документ насчитывает около тысячи страниц, и неподготовленному читателю имеет смысл ознакомиться в нем лишь с первым и последним разделами (краткий обзор и словарь терминов). Зато, если человек уже знаком с *UML*, изучение метамодели языка - весьма интересное и полезное занятие.

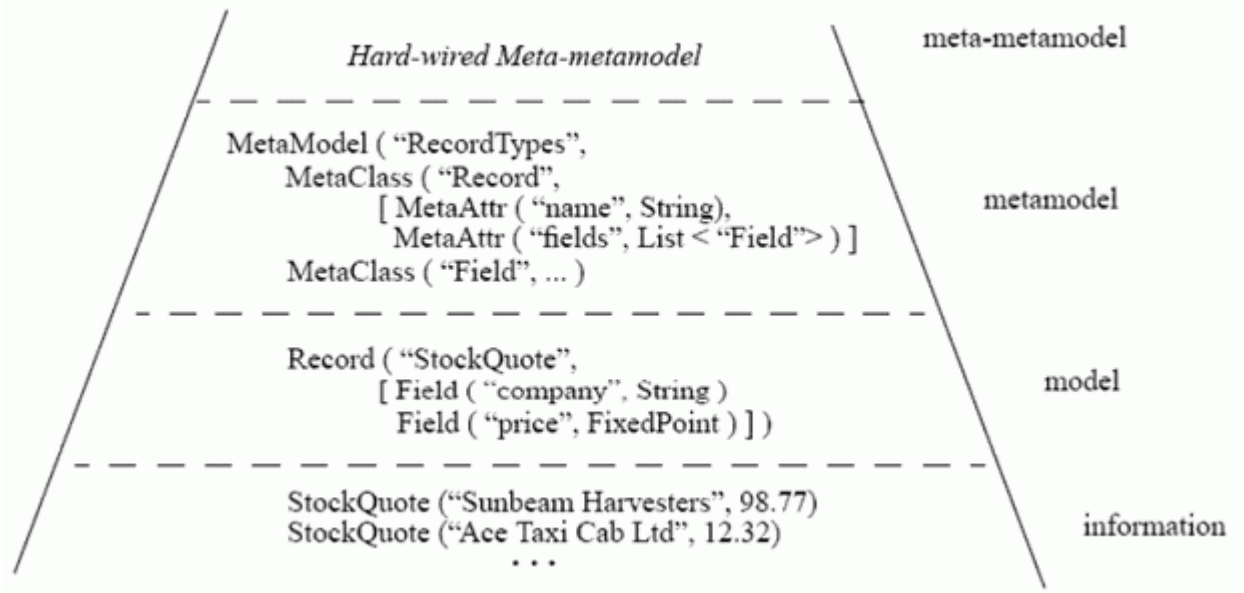


Рисунок 1.4.

Терминология и нотация

Вопрос терминологии в программной инженерии, а тем более РУССКОЙ (не говоря уже об украинской) терминологии, - вопрос сложный. Дело в том, что оригинальная терминология *UML* не всегда последовательна и довольно запутана. Русская же терминология еще не успела сложиться, ведь *UML* как технология проектирования сама *по* себе очень молода, да и русскоязычная литература *по* нему стала появляться, как всегда, с некоторым опозданием. Некоторые авторы пытаются каждый термин передать "осмысленными", "хорошими русскими словами", что не всегда удается. С точки зрения автора, искать русские аналоги уже привычных английских терминов - занятие ненужное и даже вредное: вспомните, как

трудно было вам найти нужную команду в *меню* русского MS Office, если вы привыкли пользоваться английским (в таких случаях родной язык сильно замедляет работу). Поэтому, наверное, проще использовать транскрипцию и не изобретать велосипед! В конце концов, хорошие английские слова (даже записанные русскими буквами) так же хороши, как и хорошие русские!

Теперь давайте поговорим о нотации. "*Нотация*" - это то, что в других языках называют "синтаксисом". Само *слово* "*нотация*" подчеркивает, что *UML* - язык графический и модели (а точнее диаграммы) не "записывают", а рисуют. Как уже говорилось выше, одна из задач *UML* - служить средством коммуникации внутри команды и при общении с заказчиком. "В рабочем порядке" диаграммы часто рисуют на бумаге от руки, причем обычно - не слишком аккуратно. Поэтому при выборе элементов нотации основным принципом был отбор значков, которые хорошо смотрелись бы и были бы правильно интерпретированы в любом случае - будь они нарисованы карандашом на салфетке или созданы на компьютере и распечатаны на лазерном принтере.

Вообще же, в *UML* используется четыре вида элементов нотации:

- фигуры,
- линии,
- значки,
- надписи.

Разберем все *по* порядку. Фигуры используются "плоские" - прямоугольники, эллипсы, ромбы и т. д. Но есть одно *исключение* - как мы увидим далее, на диаграмме развертывания для обозначения узлов инфраструктуры применяется "трехмерное" изображение параллелепипеда. Это единственное *исключение* из правил. Внутри любой фигуры могут помещаться другие элементы нотации.

О линиях стоит сказать лишь то, что своими концами они должны соединяться с фигурами. На *UML* диаграммах вы не встретите линий, нарисованных "сами *по* себе" и не соединяющих фигуры. Применяется два типа линий - сплошная и пунктирная. Линии могут пересекаться, и хотя таких случаев следует *по* возможности избегать, в этом нет ничего страшного.

Вообще же стоит сказать, что *UML* предоставляет исключительную свободу - можно рисовать что угодно и как вздумается, лишь бы можно было понять смысл созданных диаграмм. В изображении фигур и значков тоже нет каких-то жестких требований, и разработчики CASE-средств для *UML*-проектирования вовсю используют эту свободу, применяя различные стили рисования, заливку фигур цветом, тени и т. д. Иногда это смотрится весьма симпатично, а иногда даже раздражает.

Кстати об инструментах рисования. Мы уже упоминали, что такое *ПО* существует, и далее мы рассмотрим этот вопрос более подробно (проведя сравнительные исследования), пока же скажем лишь о нескольких наиболее заметных программах этого класса. К таким пакетам можно отнести:

- IBM Rational Rose;
- Borland Together;
- Gentleware Poseidon;
- Microsoft Visio;
- Telelogic TAU G2.

Наиболее известными из этой пятерки являются Rational Rose и Together. Это действительно средства для проектирования, а не рисования, как Visio. Долгое время *автор* этих строк использовал Poseidon, благо имеется бесплатная Community edition-версия этого продукта. Так было до тех пор, пока на одной из конференций *по* программной инженерии он не увидел TAU G2 от Telelogic. О TAU все слышали, но никто его не видел. Это легендарное средство моделирования, которое сочетает в себе мощь и простоту использования, предоставляя уникальную возможность начальной верификации моделей. И хотя *интерфейс* TAU выглядит несколько аскетично, его возможности и удобство работы просто потрясают. Все диаграммы в этом курсе созданы именно с использованием TAU, любезно предоставленным фирмой Telelogic (см. http://www.telelogic.com/).

Сейчас немного не к месту об этом говорить, но хочется упомянуть еще об одном чудесном продукте, который очень помог нам в написании этого курса. Это Zicom Mentor от Sparx Systems, выпустившего *Enterprise Architect* (см. http://www.sparxsystems.com.au/). Zicom Mentor - это простая и понятная *утилита*, представляющая собой словарь/ассистент *по UML*2.0. Zicom Mentor ответит на ваши вопросы, поможет получить и проверить ваши знания, начать новый проект. Zicom Mentor включает интерактивные курсы, электронные книги и тесты и множество другой справочной информации *по UML*.

Но давайте вернемся к нашему разговору. Как уже было сказано выше, *UML*-модель состоит из совокупности диаграмм. *UML*-диаграммы бывают различных видов, о многих из которых мы поговорим далее.

Выводы

1. UML - еще один формальный язык, который необходимо освоить каждому, кто собирается заниматься программной инженерией.
2. Само собой разумеется, что знание UML не гарантирует построения разумных и понятных моделей, хотя и является для этого необходимым.

3. UML предоставляет огромную свободу при рисовании диаграмм и выборе инструмента рисования. Производители инструментов также воспользовались этой свободой, чтобы по своему разумению "украсить" имеющуюся нотацию.

Контрольные вопросы

1. Как расшифровывается аббревиатура UML?
2. Какая версия UML является текущей?
3. Кто были авторами UML?
4. Чем НЕ является UML?
5. Какие программные средства, поддерживающие UML, вы знаете?
6. Используются ли в UML "трехмерные" фигуры?

Занятие 2. Виды диаграмм UML

Назначение диаграмм

Прежде чем перейти к обсуждению основного материала, давайте поговорим о том, зачем вообще строить какие-то диаграммы. Разработка модели любой системы (не только программной) всегда предшествует ее созданию или обновлению. Это необходимо хотя бы для того, чтобы яснее представить себе решаемую задачу. Продуманные модели очень важны и для взаимодействия внутри команды разработчиков, и для взаимопонимания с заказчиком. В конце концов, это позволяет убедиться в "архитектурной согласованности" проекта до того, как он будет реализован в коде.

Мы строим модели сложных систем, потому что не можем описать их полностью, "окинуть одним взглядом". Поэтому мы выделяем лишь существенные для конкретной задачи свойства системы и строим ее модель, отображающую эти свойства. Метод объектно-ориентированного анализа позволяет описывать реальные сложные системы наиболее адекватным образом. Но с увеличением сложности систем возникает потребность в хорошей технологии моделирования. Как мы уже говорили ранее, в качестве такой "стандартной" технологии используется унифицированный *язык моделирования* (*Unified Modeling Language, UML*), который является графическим языком для спецификации, визуализации, проектирования и документирования систем. С помощью *UML* можно разработать подробную модель создаваемой системы, отображающую не только ее концепцию, но и конкретные особенности реализации. В рамках *UML*-модели все представления о системе фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Примечание. Мы рассмотрим не все, а лишь некоторые из видов диаграмм. Например, диаграмма компонентов не рассматривается, это лишь краткий обзор видов диаграмм. Количество типов диаграмм для конкретной модели приложения никак не ограничивается. Для простых приложений нет необходимости строить диаграммы всех без исключения типов. Некоторые из них могут просто отсутствовать, и этот факт не будет считаться ошибкой. Важно понимать, что наличие диаграмм определенного вида зависит от специфики конкретного проекта. Информацию о других (не рассмотренных здесь) видах диаграмм можно найти в стандарте UML.

Почему нужно несколько видов диаграмм

Для начала определимся с терминологией. В предисловии мы неоднократно использовали понятия системы, модели и диаграммы. *Автор* уверен, что каждый из нас интуитивно понимает смысл этих понятий, но, чтобы внести полную *ясность*, снова заглянем в *глоссарий* и прочтем следующее:

Система - совокупность взаимосвязанных управляемых подсистем, объединенных общей целью функционирования.

Да, не слишком информативно. А что же такое тогда подсистема? Чтобы прояснить ситуацию, обратимся к классикам:

Системой называют набор подсистем, организованных для достижения определенной цели и описываемых с помощью совокупности моделей, возможно, с различных точек зрения.

Что ж, ничего не попишешь, придется искать *определение* подсистемы. Там же сказано, что **подсистема** - это совокупность элементов, часть из которых задает спецификацию поведения других элементов. Ян Соммервилл объясняет это понятие таким образом:

Подсистема - это система, функционирование которой не зависит от сервисов других подсистем. Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами.

Тоже не слишком понятно, но уже лучше. Говоря "человеческим" языком, система представляется в виде набора более простых сущностей, которые относительно самодостаточны. Это можно сравнить с тем, как в процессе разработки программы мы строим графический *интерфейс* из стандартных "кубиков" - визуальных компонентов, или как сам текст программы тоже разбивается на модули, которые содержат подпрограммы, объединенные *по* функциональному признаку, и их можно использовать повторно, в следующих программах.

С понятием системы разобрались. В процессе проектирования система рассматривается **с разных точек зрения** с помощью моделей, различные представления которых предстают в форме диаграмм. Опять-таки у читателя могут возникнуть вопросы о смысле понятий *модели* и *диаграммы*. Думаем, красивое, но не слишком понятное *определение* **модели как семантически замкнутой абстракции системы** вряд ли прояснит ситуацию, поэтому попробуем объяснить "своими словами".

Модель - это некий (материальный или нет) *объект*, отображающий лишь наиболее значимые для данной задачи характеристики системы. Модели бывают разные - материальные и нематериальные, искусственные и естественные, декоративные и математические...

Приведем несколько примеров. Знакомые всем нам пластмассовые игрушечные автомобильчики, которыми мы с таким азартом играли в детстве, это не что иное, как *материальная искусственная декоративная* модель реального автомобиля. Конечно, в таком "авто" нет двигателя, мы не заполняем его бак бензином, в нем не работает (более того, вообще отсутствует) коробка передач, но как модель эта игрушка свои функции вполне выполняет:

она дает ребенку *представление* об автомобиле, поскольку отображает его характерные черты - наличие четырех колес, кузова, дверей, окон, способность ехать и т. д.

В ходе медицинских исследований опыты на животных часто предшествуют клиническим испытаниям медицинских препаратов на людях. В таком случае животное выступает в роли *материальной естественной* модели человека.

$$mg - a\frac{dx}{dt} = m\frac{d^2x}{dt^2}$$

Уравнение, изображенное выше - тоже модель, но это модель математическая, и описывает она движение материальной точки под действием силы тяжести.

Осталось лишь сказать, что такое *диаграмма*.

Диаграмма - это графическое *представление множества* элементов. Обычно изображается в виде графа с вершинами (сущностями) и ребрами (отношениями). Примеров диаграмм можно привести множество. Это и знакомая нам всем со школьных лет *блок-схема*, и схемы монтажа различного оборудования, которые мы можем видеть в руководствах пользователя, и *дерево* файлов и каталогов на диске, которое мы можем увидеть, выполнив в консоли *Windows* команду tree, и многое-многое другое. В повседневной жизни диаграммы окружают нас со всех сторон, ведь рисунок воспринимается нами легче, чем текст...

Но вернемся к проектированию *ПО* (и не только). В этой отрасли с **помощью диаграмм можно визуализировать систему с различных точек зрения**. Одна из диаграмм, например, может описывать взаимодействие пользователя с системой, другая - изменение состояний системы в процессе ее работы, третья - взаимодействие между собой элементов системы и т. д. Сложную систему можно и нужно представить в виде набора небольших и почти независимых моделей-диаграмм, причем ни одна из них не является достаточной для описания системы и получения полного представления о ней, поскольку каждая из них фокусируется на каком-то определенном аспекте функционирования системы и выражает разный *уровень абстракции*. Другими словами, каждая модель соответствует некоторой определенной, частной точке зрения на проектируемую систему.

Несмотря на то что в предыдущем абзаце мы весьма вольготно обошлись с понятием модели, следует понимать, что в контексте приведенных выше определений **ни одна отдельная диаграмма не является моделью**. Диаграммы - лишь средство визуализации модели, и эти два понятия следует различать. Лишь **набор диаграмм составляет модель системы**и наиболее полно ее описывает, но не одна *диаграмма*, вырванная из контекста.

Виды диаграмм

UML 1.5 определял **двенадцать типов диаграмм**, разделенных на три группы:

- четыре типа диаграмм представляют статическую структуру приложения;

- пять представляют поведенческие аспекты системы;
- три представляют физические аспекты функционирования системы (диаграммы реализации).

Текущая версия *UML* 2.1 внесла не слишком много изменений. Диаграммы слегка изменились внешне (появились фреймы и другие визуальные улучшения), немного усовершенствовалась *нотация*, некоторые диаграммы получили новые наименования.

Впрочем, *точное число канонических диаграмм* для нас абсолютно неважно, так как мы рассмотрим не все из них, а лишь некоторые - *по* той причине, что количество типов диаграмм для конкретной модели конкретного приложения не является строго фиксированным. Для простых приложений нет необходимости строить все без исключения диаграммы. Например, для локального приложения не обязательно строить диаграмму развертывания. Важно понимать, что перечень диаграмм зависит от специфики разрабатываемого проекта и определяется самим разработчиком. Если же любопытный читатель все-таки пожелает узнать обо всех диаграммах *UML*, мы отошлем его к стандарту *UML*(http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML). Напомним, что цель этого курса - не описать абсолютно все возможности *UML*, а лишь познакомить с этим языком, дать первоначальное *представление* об этой технологии.

Итак, мы кратко рассмотрим такие виды диаграмм, как:

- диаграмма прецедентов;
- диаграмма классов;
- диаграмма объектов;
- диаграмма последовательностей;
- диаграмма взаимодействия;
- диаграмма состояний;
- диаграмма активности;
- диаграмма развертывания.

О некоторых из этих диаграмм мы будем говорить подробнее на следующих занятиях. Пока же мы не станем заострять внимание на подробностях, а зададимся целью научить читателя хотя бы визуально различать виды диаграмм, дать начальное *представление* о назначении основных видов диаграмм.

Диаграмма прецедентов (use case diagram)

Любые (в том числе и программные) системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются **экторами**, причем каждый эктор ожидает, что система будет вести себя строго

определенным, предсказуемым образом. Попробуем дать более строгое определение эктора. Для этого воспользуемся замечательным визуальным словарем по UML *Zicom Mentor*:

Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо " *человечком* ", подобным тем, которые мы рисовали в детстве, изображая членов своей семьи, либо *символом класса с соответствующим стереотипом*, как показано на рисунке. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить (рисунок 2.1).

Внимательный читатель сразу же может задать вопрос: *а почему эктор, а не актер*? Согласны, слово "эктор" немного режет слух русского человека. Причина же, почему мы говорим именно так, проста - эктор образовано от слова **action**, что в переводе означает *действие*. Дословный же перевод слова "эктор" - *действующее лицо* - слишком длинный и неудобный для употребления. Поэтому мы будем и далее говорить именно так.

Рисунок 2.1

Тот же внимательный читатель мог заметить промелькнувшее в определении эктора слово "прецедент". Что же это такое? Этот вопрос заинтересует нас еще больше, если вспомнить, что сейчас мы говорим о *диаграмме прецедентов*. Итак,

Прецедент (use-case) - описание отдельного аспекта поведения системы с точки зрения пользователя (Буч).

Определение вполне понятное и исчерпывающее, но его можно еще немного уточнить, воспользовавшись тем же *Zicom Mentor* '.

Прецедент (use case) - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. *Прецеденты и экторы соединяются с помощью линий*. Часто на одном из концов линии изображают *стрелку*, причем *направлена она к тому, у кого запрашивают сервис*, другими словами, чьими услугами пользуются. Это простое объяснение иллюстрирует *понимание прецедентов как сервисов*, пропагандируемое компанией IBM.

Рисунок 2.2.

Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. *Все эти возможности мы здесь рассматривать не будем*. Как уже говорилось выше, цель этого обзора - просто научить читателя выделять диаграмму прецедентов, понимать ее назначение и смысл обозначений, которые на ней встречаются.

Кстати, к этому моменту мы уже потратили достаточно много времени на объяснение понятий и их условных обозначений. Наверное, пора уже, наконец, привести пример диаграммы прецедентов. Как вы думаете, что означает эта диаграмма (рисунок 2.3)?

Рисунок 2.3.

Полагаем, здесь все было бы понятно, если бы даже мы никогда не слышали о диаграммах прецедентов! Ведь так? Все мы в студенческие годы пользовались библиотеками (которые теперь для нас заменил Интернет), и потому все это для нас знакомо. Обратите также внимание на примечание, сопоставленное с одним из прецедентов. Следует заметить, что иногда на диаграммах прецедентов *границы системы обозначают прямоугольником*, в верхней части которого может быть указано *название системы*. Таким образом, прецеденты

- действия, выполняемые системой в ответ на действия эктора, - помещаются внутри прямоугольника.

А вот еще один пример (рисунок 2.4). Думаем, вы сами, без нашей помощи, легко догадаетесь, о чем там идет речь.

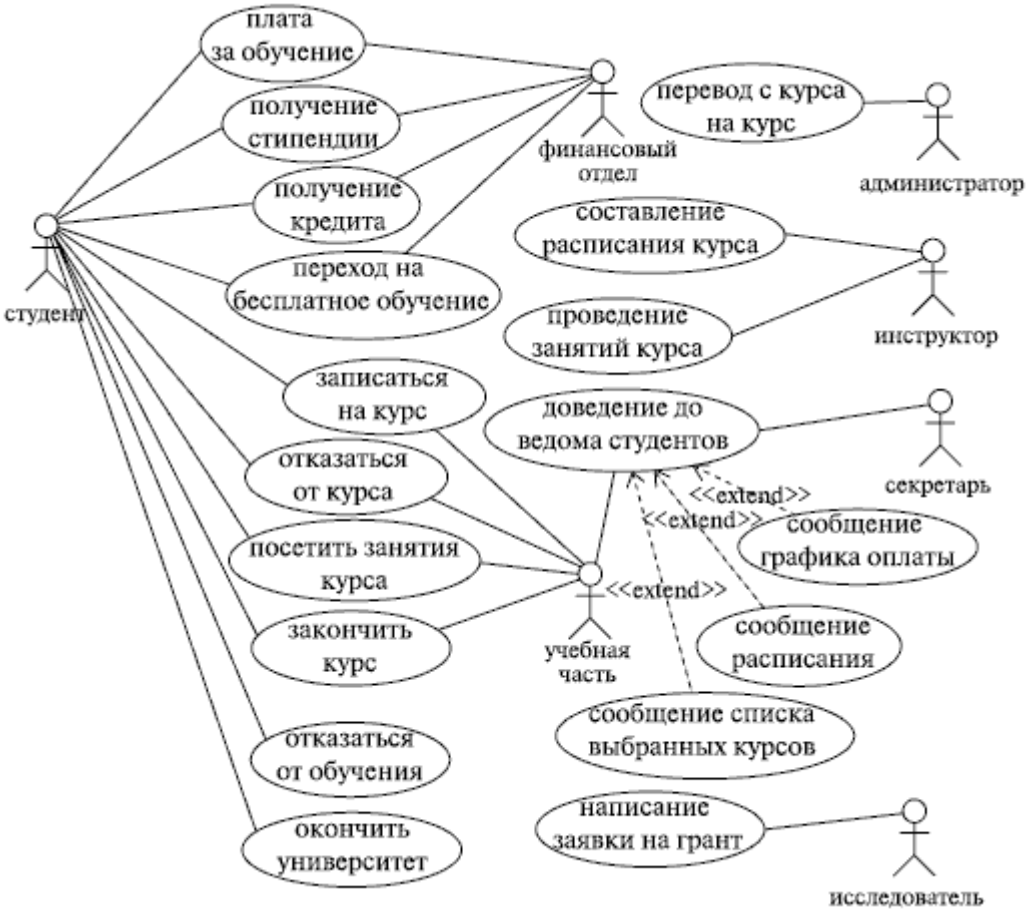


Рисунок 2.4.

Из всего сказанного выше становится понятно, что *диаграммы прецедентов* относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное *средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями* продукта. Как мы уже могли убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Подводя итоги, можно выделить такие **цели создания диаграмм прецедентов**:

- определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
- формирование общих требований к поведению проектируемой системы;
- разработка концептуальной модели системы для ее последующей детализации;
- подготовка документации для взаимодействия с заказчиками и пользователями системы.

Диаграмма классов (class diagram)

Вообще-то, понятие класса нам уже знакомо, но, пожалуй, не лишним будет поговорить о классах еще раз. Классики о классах говорят очень просто и понятно:

Класс (class) - категория вещей, которые имеют общие атрибуты и операции.

Продолжая тему, скажем, что классы - *это строительные блоки любой объектно-ориентированной системы*. Они представляют собой описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. При проектировании объектно-ориентированных систем диаграммы классов обязательны.

Классы используются в процессе анализа предметной области для составления *словаря предметной области* разрабатываемой системы. Это могут быть как абстрактные понятия предметной области, так и классы, на которые опирается разработка и которые описывают программные или аппаратные сущности.

Диаграмма классов - это *набор статических*, декларативных *элементов модели*. Диаграммы классов могут применяться и при прямом проектировании, то есть в процессе разработки новой системы, и при *обратном проектировании* - описании существующих и используемых систем.

Информация с диаграммы классов напрямую отображается в исходный код приложения - в большинстве существующих инструментов UML-моделирования возможна *кодогенерация* для определенного языка программирования (обычно Java или C++). Таким образом, диаграмма классов - конечный результат проектирования и отправная точка процесса разработки.

Но мы опять заговорились, а, как известно, лучше один раз увидеть, чем сто раз услышать. Мы уже знаем, как классы обозначаются в UML, но пока еще не видели ни одной диаграммы "с их участием". Итак, посмотрим на примеры диаграмм классов.

Первый пример (рисунок 2.5) весьма прост.

Как видим, он, хоть и немного однобоко, иллюстрирует с помощью операции наследования или генерализации "генеалогическое древо" бытовой техники. Думаем, мы бы поняли смысл этой диаграммы, даже если бы ничего не знали о классах и не занимались программированием вообще.

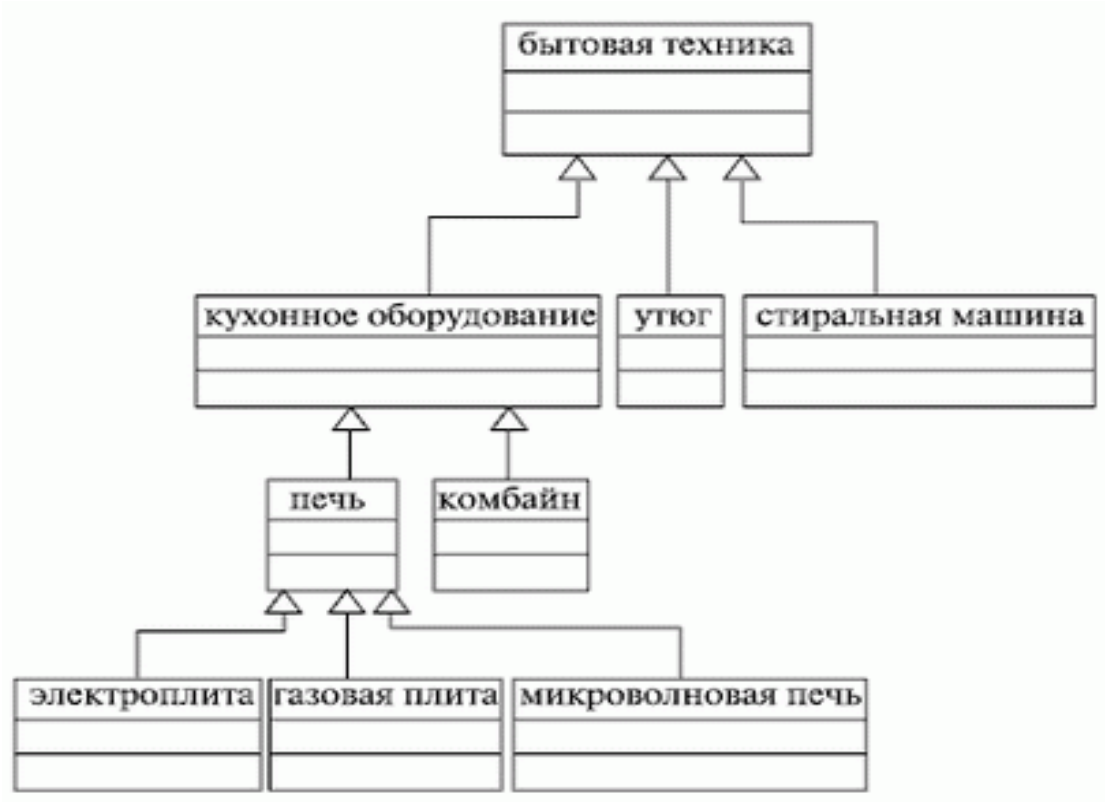


Рисунок 2.5.

Рассмотрим еще пример (рисунок 2.6):

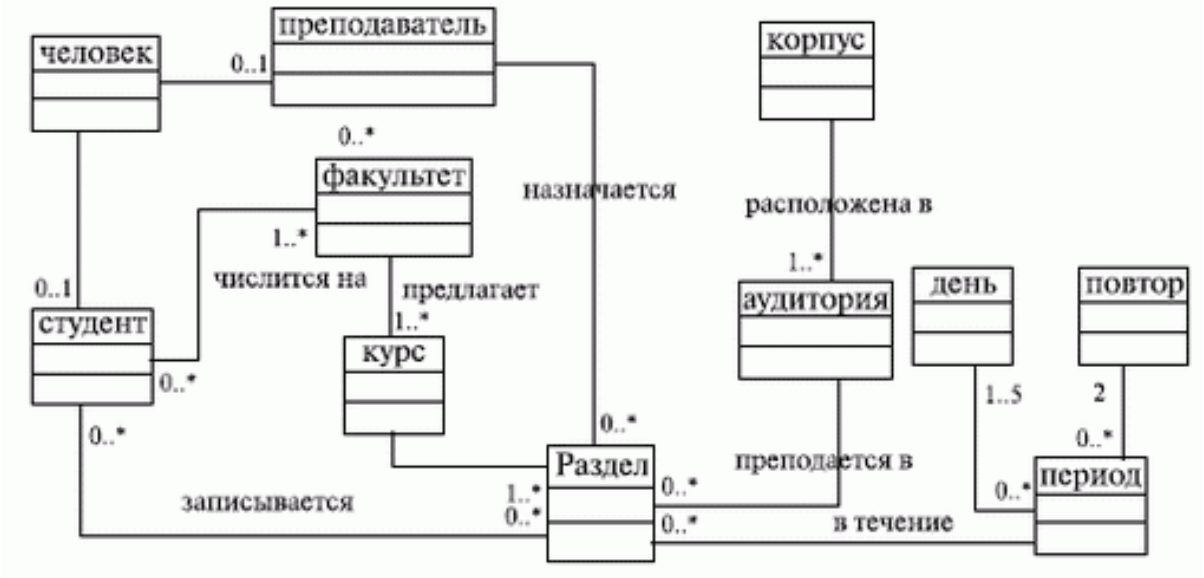


Рисунок 2.6.

И опять-таки смысл этой диаграммы ясен без особых пояснений. Даже бегло рассмотрев ее, можно легко догадаться, что она описывает предметную область задачи об автоматизации работы некоего вуза или учебного центра. Обратите внимание на обозначения кратности на концах связей. А теперь немного усложним задачу (рисунок 2.7).

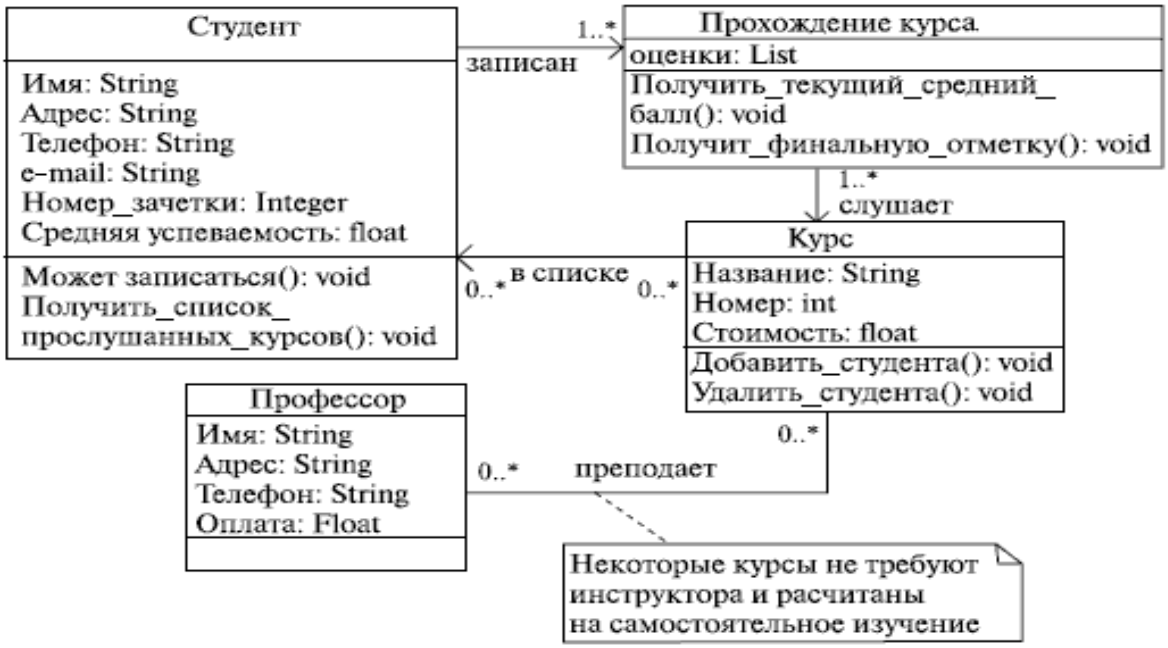


Рисунок 2.7.

Как видим, здесь уже все более серьезно - кроме кратности обозначены свойства (и их типы) и операции, и вообще, эта диаграмма производит впечатление набора классов для реализации, а не просто описания предметной области, как предыдущие. Но, тем не менее, все равно все просто и понятно.

Отметим, что более детально о диаграмме классов мы поговорим далее. Там мы подробно разберем нотацию этого вида диаграмм и познакомимся с улучшениями, внесенными текущей версией UML.

Диаграмма объектов (object diagram)

И снова, прежде чем говорить о новом виде диаграмм, введем определения нужных нам понятий. Итак, мы уже знаем, что такое класс. А что такое объект? Обратимся к классикам, которые об объектах говорят так же просто и понятно, как и о классах:

Объект (object) - экземпляр класса.

Zicom Mentor "говорит" об объектах более обстоятельно:

Объект (object) -

- конкретная материализация абстракции;
- сущность с хорошо определенными границами, в которой инкапсулированы состояние и поведение;
- экземпляр класса (вернее, классификатора - эктор, класс или интерфейс). Объект уникально идентифицируется значениями атрибутов, определяющими его состояние в данный момент времени.

"Второе" определение, по сути, просто расширяет "Бучевское". Да, действительно, объект - это экземпляр класса. Скажем, объектом класса "Микроволновая печь" из примера,

приведенного выше, может быть и простейший прибор фирмы "*Saturn*" небольшой емкости и с механическим управлением, и навороченный агрегат с грилем, сенсорным управлением и системой трехмерного распределения энергии от Samsung или LG.

Еще пример - все мы являемся объектами класса "человек" и различимы между собой по таким признакам (значениям атрибутов), как имя, цвет волос, глаз, рост, вес, возраст и т. д. (в зависимости от того, какую задачу мы рассматриваем и какие свойства человека для нас в ней важны).

Как же обозначается объект в UML? А очень просто - объект, *как и класс, обозначается прямоугольником, но его имя подчеркивается*. Под словом имя здесь мы понимаем название объекта и наименование его класса, разделенные двоеточием. Для указания значений атрибутов объекта в его обозначении может быть предусмотрена специальная секция. Еще один нюанс состоит в том, что объект может быть анонимным: это нужно в том случае, если в данный момент не важно, какой именно объект данного класса принимает участие во взаимодействии. Примеры - на рисунок 2.8.

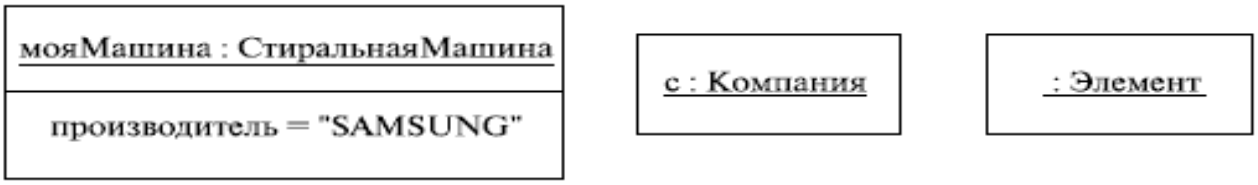


Рисунок 2.8.

Итак, на определение основных понятий мы потратили довольно много времени, и пора бы уже вернуться к основному предмету нашего внимания - **диаграмме объектов**. Для чего нужны *диаграммы объектов*? Они показывают множество объектов - экземпляров классов (изображенных на диаграмме классов) и отношений между ними в некоторый момент времени. То есть *диаграмма объектов - это своего рода снимок состояния системы в определенный момент времени*, показывающий множество объектов, их состояния и отношения между ними в данный момент.

Таким образом, диаграммы объектов представляют статический вид системы с точки зрения проектирования и процессов, являясь основой для сценариев, описываемых диаграммами взаимодействия. Говоря другими словами, диаграмма объектов используется для пояснения и детализации диаграмм взаимодействия, например, диаграмм последовательностей. Впрочем, авторам курса очень редко доводилось применять этот тип диаграмм.

Приведем простейший пример такой диаграммы (рисунок 2.9).

Рисунок 2.9.

О чем здесь идет речь, в принципе, понятно: некоторая фирма "раскручивает" новый товар или услугу. В этом процессе участвуют вице-президент по маркетингу, вице-президент по продажам, менеджер по продажам, торговый агент, специалист по рекламе, некое печатное издание и покупатель. Причем даже без указания сообщений, которыми обмениваются эти объекты, отлично видно, кто с кем взаимодействует. Кстати, обратите внимание, что на этой диаграмме все объекты анонимные!

Другой пример (рисунок 2.10).

Рисунок 2.10.

Эта диаграмма тоже понятна в общих чертах даже без дополнительных объяснений. Здесь мы видим взаимосвязь объектов - организационных единиц в некоторой компании.

И наконец, последний пример (рисунок 2.11): *диаграмма объектов* учебной среды "Робот" для Turbo Pascal, в которой наше поколение школьников училось основам алгоритмизации.

Думаем, пока примеров достаточно и главной цели мы достигли - научили читателя различать диаграмму объектов. Кому-то может показаться, что мы уделили ей мало внимания, но, как уже было сказано выше, читатель вряд ли будет часто встречаться с этим типом диаграмм.

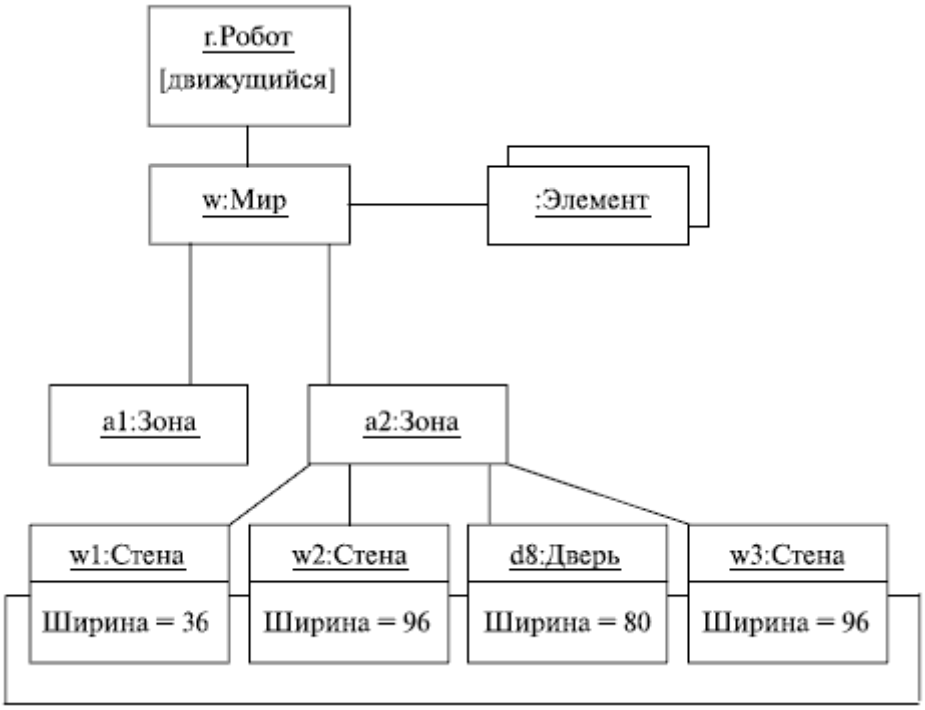


Рисунок 2.11.

Диаграмма последовательностей (sequence diagram)

Только что мы познакомились с диаграммой объектов, которая показывает отношения между объектами в некоторый момент времени, т. е. предоставляет нам снимок состояния системы, являясь статической. **Диаграмма же последовательностей** *отображает взаимодействие объектов в динамике*. Что значит "в динамике"? Как раз с этим нам и предстоит разобраться.

В UML взаимодействие объектов понимается как обмен информацией между ними. При этом информация принимает вид сообщений. Кроме того, *что сообщение несет какую-то информацию, оно некоторым образом также влияет на получателя*. Как видим, в этом плане *UML полностью соответствует основным принципам ООП*, в соответствии с которыми информационное взаимодействие между объектами сводится к отправке и приему сообщений.

Диаграмма последовательностей относится к диаграммам взаимодействия UML, описывающим поведенческие аспекты системы, но рассматривает взаимодействие объектов во времени. Другими словами, диаграмма последовательностей отображает временные особенности передачи и приема сообщений объектами.

Искушенный читатель, возможно, скажет, что нечто подобное делает и *диаграмма прецедентов*. Да, действительно, *диаграммы последовательностей можно (и нужно!) использовать для уточнения диаграмм прецедентов*, более детального описания логики сценариев использования. Это отличное средство документирования проекта с точки зрения сценариев использования! Диаграммы последовательностей обычно содержат *объекты*, которые взаимодействуют в рамках сценария, *сообщения*, которыми они обмениваются, и *возвращаемые результаты*, связанные с сообщениями. Впрочем, часто возвращаемые результаты обозначают лишь в том случае, если это не очевидно из контекста.

Теперь о том, какие обозначения используются на диаграмме последовательностей. Как и ранее, объекты обозначаются прямоугольниками с подчеркнутыми именами (чтобы отличить их от классов), сообщения (вызовы методов) - линиями со стрелками, возвращаемые результаты - пунктирными линиями со стрелками. Прямоугольники на вертикальных линиях под каждым из объектов показывают "время жизни" (фокус) объектов. Впрочем, довольно часто их не изображают на диаграмме, все это зависит от индивидуального стиля проектирования.

Поскольку текст предыдущего абзаца, может быть, не слишком хорошо воспринимается на слух, да и лучше, как известно, "один раз увидеть, чем сто раз услышать", *приведем пример диаграммы последовательностей* (рисунок 2.12):

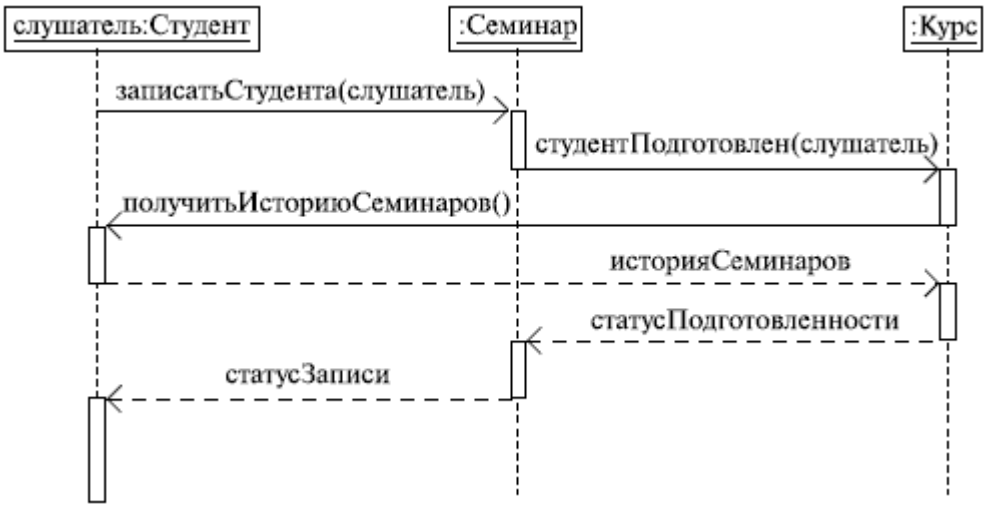


Рисунок 2.12.

Думаем, смысл диаграммы вполне понятен: студент хочет записаться на некий семинар, предлагаемый в рамках некоторого учебного курса. С этой целью проводится проверка подготовленности студента, для чего запрашивается список (история) семинаров курса, уже пройденных студентом (перейти к следующему семинару можно, лишь проработав материал предыдущих занятий - знакомая картина, не правда ли?). После получения истории семина-

ров объект класса "Слушатель" получает статус подготовленности, на основе которой студенту сообщается результат (статус) его попытки записи на семинар. Кстати, обратите внимание на вызов методов. Как видите, все просто!

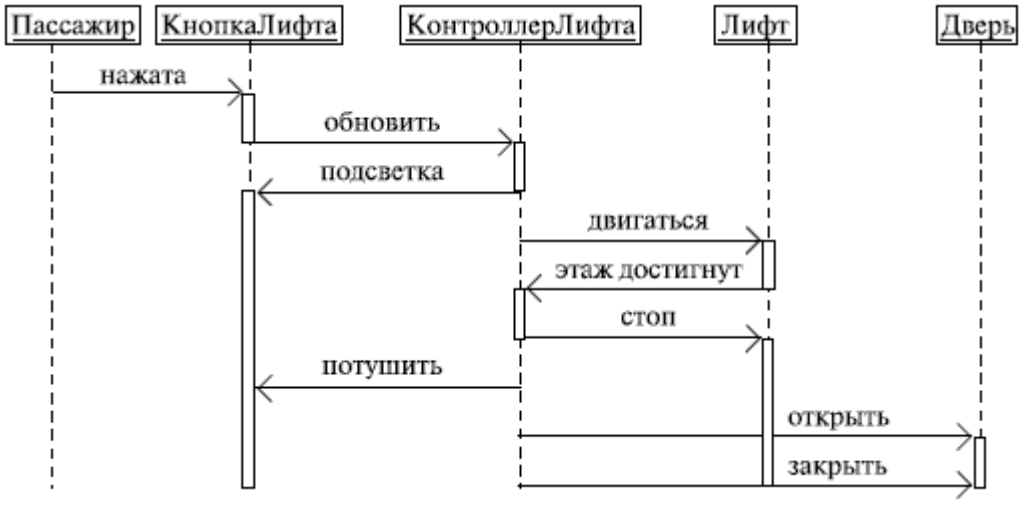


Рисунок 2.13.

И наконец, *еще один пример* (рисунок 2.14).

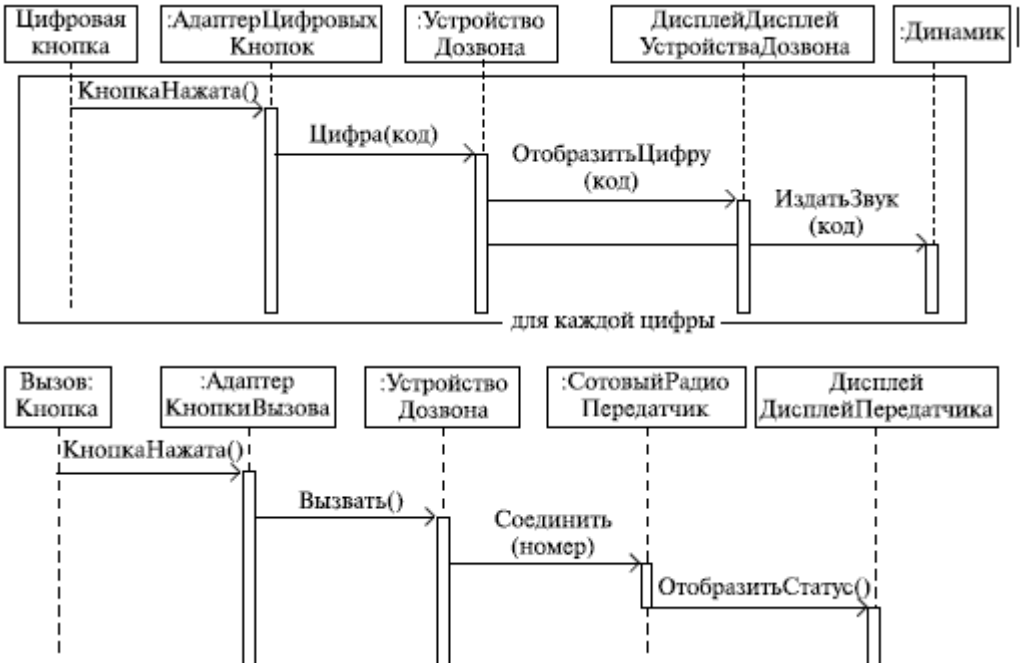


Рисунок 2.14.

Узнаете свой мобильный?

Диаграмма взаимодействия (кооперации, collaboration diagram)

Диаграммы последовательностей - это отличное средство документирования поведения системы, детализации логики сценариев использования; но есть еще один способ - использовать диаграммы взаимодействия. Диаграмма взаимодействия *показывает поток сообщений между объектами системы и основные ассоциации между ними* и по сути, как уже

было сказано выше, является альтернативой диаграммы последовательностей. Внимательный читатель, возможно, скажет, что *диаграмма объектов* делает то же самое, - и будет не прав. *Диаграмма объектов показывает статику*, некий снимок системы, связи между объектами в данный момент времени, диаграмма же взаимодействия, как и диаграмма последовательностей, показывает взаимодействие (извините за невольный каламбур) объектов во времени, т. е. в динамике.

Следует отметить, что использование диаграммы последовательностей или диаграммы взаимодействия - личный выбор каждого проектировщика и зависит от индивидуального стиля проектирования. Мы, например, чаще отдаем предпочтение диаграмме последовательностей. На обозначениях, применяемых на диаграмме взаимодействия, думаем, не стоит останавливаться подробно. Здесь все стандартно: объекты обозначаются прямоугольниками с подчеркнутыми именами (чтобы отличить их от классов, помните?), ассоциации между объектами указываются в виде соединяющих их линий, над ними может быть изображена стрелка с указанием названия сообщения и его порядкового номера.

Необходимость номера сообщения объясняется очень просто - в отличие от диаграммы последовательностей, *время на диаграмме взаимодействия не показывается в виде отдельного измерения*. Поэтому последовательность передачи сообщений можно указать только с помощью их нумерации. В этом и состоит вероятная причина пренебрежения этим видом диаграмм многими проектировщиками.

Но давайте же, наконец, перейдем к примерам (рисунок 2.15).

Рисунок 2.15.

Как видите, эта диаграмма описывает (очень грубо) работу персонала библиотеки по обслуживанию клиентов: библиотекарь получает заказ от клиента, поручает сотруднику найти информацию по нужной клиенту книге, а после получения данных поручает еще одному сотруднику выдать книгу клиенту. Разобрались? Тогда еще пример (рисунок 2.16).

Рисунок 2.16.

Надеемся, что и эта диаграмма не смогла поставить вас в тупик. Скорее всего, она описывает процесс управления учебными курсами (очевидно, путем создания их из готовых модулей) для некоего учебного центра. Как видите, все просто!

И, наконец, еще один пример (рисунок 2.17), который должен вызвать легкое "дежавю" у внимательного читателя.

Рисунок 2.17.

Конечно же! Ведь это последний пример, который мы рассматривали, говоря о диаграммах последовательностей, - мобильный телефон! Как видим, это просто другая форма представления, к тому же, на наш взгляд, менее удобная. Впрочем, в команде могут работать различные люди, с различными предпочтениями и особенностями восприятия, так что какой вид диаграмм использовать для описания логики сценариев - диаграммы последовательностей или диаграммы кооперации - решать вам.

Диаграмма состояний (statechart diagram)

Объекты характеризуются поведением и состоянием, в котором находятся. Например, человек может быть новорожденным, младенцем, ребенком, подростком или взрослым.

Другими словами, объекты что-то делают и что-то "знают". **Диаграммы состояний** *применяются для того, чтобы объяснить, каким образом работают сложные объекты*. Несмотря на то что смысл понятия "состояние" интуитивно понятен, все же приведем его определение в таком виде, в каком его дают классики и Zicom Mentor:

Состояние (state) - ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает какого-то события. Состояние объекта определяется значениями некоторых его атрибутов и присутствием или отсутствием связей с другими объектами.

Диаграмма состояний показывает, как объект переходит из одного состояния в другое. Очевидно, что диаграммы состояний служат для моделирования динамических аспектов системы (как и диаграммы последовательностей, кооперации, прецедентов и, как мы увидим далее, диаграммы деятельности). Часто можно услышать, что *диаграмма состояний показывает автомат*, но об этом мы поговорим подробнее чуть позже. Диаграмма состояний полезна при *моделировании жизненного цикла* объекта (как и ее частная разновидность - диаграмма деятельности, о которой мы будем говорить далее).

От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса - одного объекта, причем объекта *реактивного*, то есть объекта, поведение которого характеризуется его реакцией на внешние события. Понятие жизненного цикла применимо как раз к реактивным объектам, настоящее состояние (и поведение) которых обусловлено их прошлым состоянием. Но диаграммы состояний важны не только для описания динамики отдельного объекта. Они могут использоваться для *конструирования исполняемых систем* путем прямого и *обратного проектирования*. И они действительно с успехом применяются в таком качестве, вспомним существующие варианты "исполняемого UML", такие как UNIMOD, FLORA и др.

Но поговорим об обозначениях на диаграммах состояний. Скругленные прямоугольники представляют состояния, через которые проходит объект в течение своего жизненного цикла. Стрелками показываются переходы между состояниями, которые вызваны выполнением методов описываемого диаграммой объекта. Существует также *два вида псевдосостояний: начальное*, в котором находится объект сразу после его создания (обозначается сплошным кружком), и *конечное*, которое объект не может покинуть, если перешел в него (обозначается кружком, обведенным окружностью).

Приведем пример простейшей диаграммы состояний (рисунок 2.18).

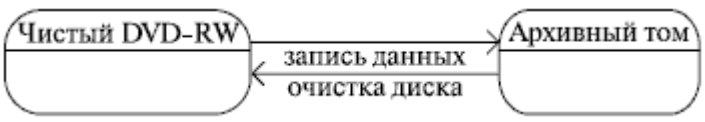


Рисунок 2.18.

Думаем, здесь все понятно без лишних слов. А вот более сложный пример (рисунок 2.19):

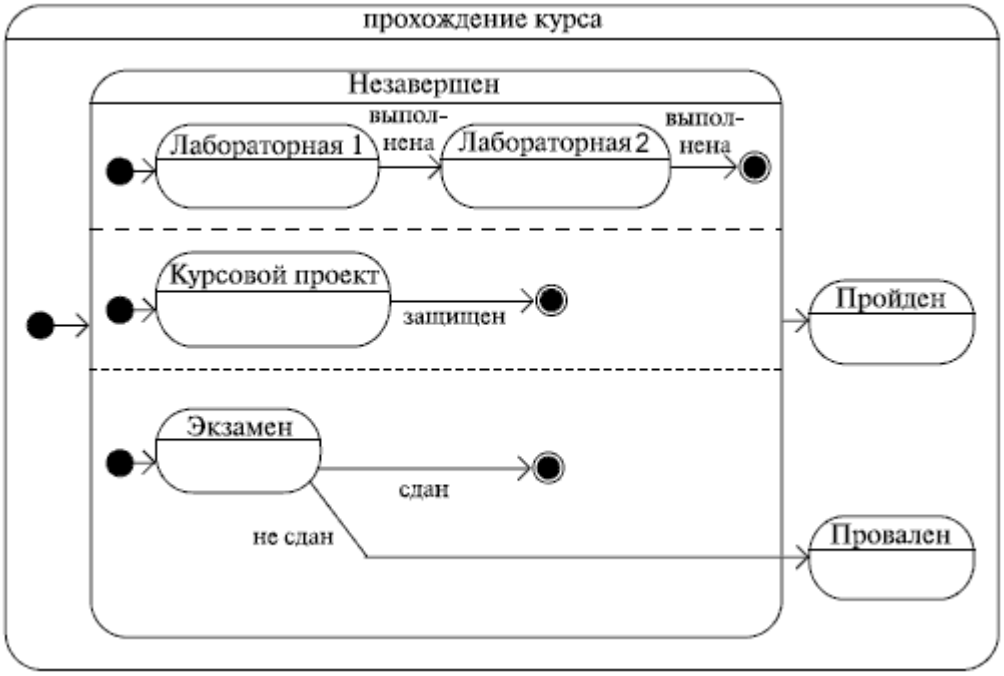


Рисунок 2.19.

Здесь мы видим *составное состояние*, включающее другие состояния, одно из которых содержит также параллельные *подсостояния*. Мы не говорили ранее о том, как все это обозначается, но ведь и так понятно - это диаграмма прохождения академического курса студентом. Для того чтобы пройти курс, студент должен выполнить лабораторные работы, защитить курсовой проект и сдать экзамен. Все просто!

А вот еще один пример (рисунок 2.20):

Рисунок 2.20.

Не сомневаемся, что вы легко догадались, что это за устройство - конечно же, таймер! Такой прибор может применяться в составе различных реле, например, для отключения телевизора по истечении указанного промежутка времени. Основное его назначение - контроль

времени (проверка, не истек ли указанный промежуток), но у него есть еще один режим работы - установка. По истечении указанного промежутка времени или при "сбросе" устройство отключается. В конце концов, о чем мы говорим - вы сами много раз устанавливали слип-таймер в телевизоре или устанавливали опцию "Выключить по завершении" в Nero *Burning* ROM!

Диаграмма активности (деятельности, activity diagram)

Когда-то на уроках информатики в школе мы рисовали блок-схемы, чтобы наглядно изобразить алгоритм решения некоторой задачи. Действительно, моделируя поведение проектируемой системы, часто недостаточно изобразить процесс смены ее состояний, а нужно также раскрыть детали алгоритмической реализации операций, выполняемых системой. Как мы уже говорили, для этой цели традиционно использовались блок-схемы или структурные схемы алгоритмов. В UML для этого существуют **диаграммы деятельности**, являющиеся частным случаем диаграмм состояний. Диаграммы деятельности удобно применять для визуализации алгоритмов, по которым работают операции классов.

Да, кстати, надеемся, вы помните, что такое алгоритм? Существует огромное количество определений этого понятия. Вот одно из них:

Алгоритм - последовательность определенных действий или элементарных операций, выполнение которых приводит к получению желаемого результата.

Алгоритмы окружают нас повсюду, хоть мы и редко задумываемся об этом. Вспомните кулинарные рецепты или руководства по эксплуатации бытовых приборов! Конечно, отечественный потребитель привык жить по принципу "если ничего не помогает, прочтите, наконец, инструкцию", но факт остается фактом: чем сложнее устройство или система, тем важнее строго следовать алгоритму.

Обозначения на диаграмме активности также напоминают те, которые мы встречали на блок-схеме, хотя есть, как мы увидим далее, и некоторые существенные отличия. С другой стороны, нотация диаграмм активности очень похожа на ту, которая используется в диаграммах состояний. Но, наверное, лучше будет просто показать пример (рисунок 2.21).

Многие из нас именно так начинают свой день, не правда ли? Обратите внимание на то, как изображено параллельное пение и принятие душа, - на обычной блок-схеме это было бы невозможно!

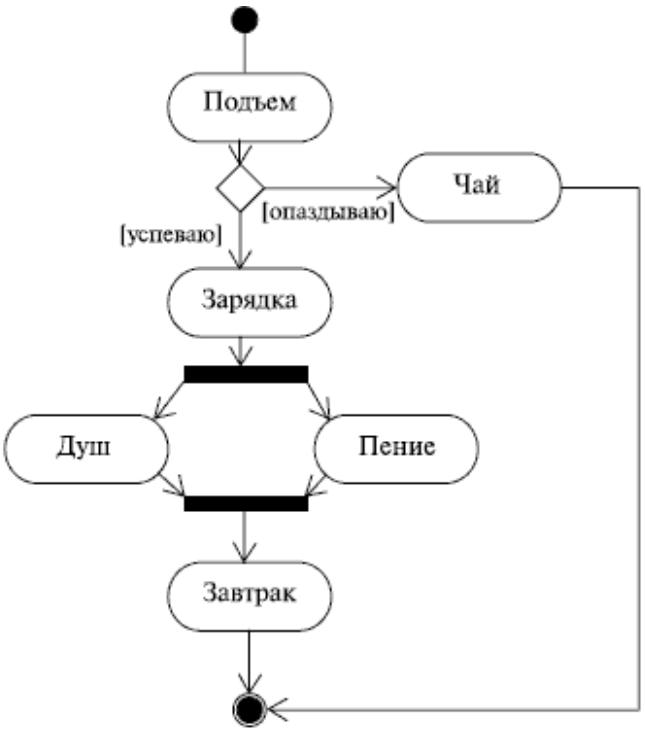


Рисунок 2.21.

А вот еще пример (рисунок 2.22):

Рисунок 2.22.

И опять все понятно - это оформление заказа в интернет-магазине! Ну и напоследок еще одна диаграмма (рисунок 2.23).

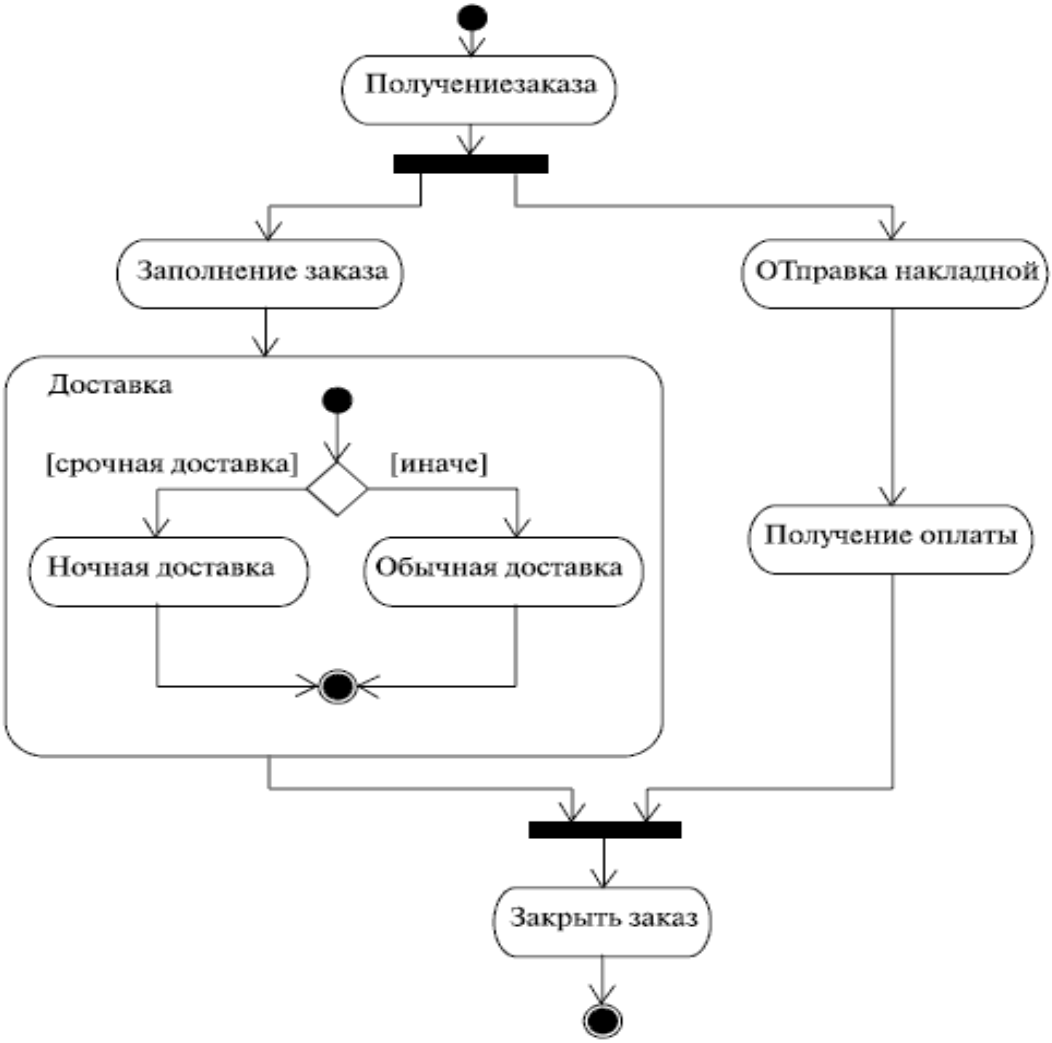


Рисунок 2.23.

Догадались, что она описывает? Сможете отличить этот тип диаграмм? Тогда пошли дальше!

Диаграмма развертывания (deployment diagram)

Когда мы пишем программу, мы пишем ее для того, чтобы запускать на компьютере, который имеет некоторую аппаратную конфигурацию и работает под управлением некоторой операционной системы. Корпоративные приложения часто требуют для своей работы некоторой *ИТ-инфраструктуры*, хранят информацию в базах данных, расположенных где-то на серверах компании, вызывают веб-сервисы, используют общие ресурсы и т. д. В таких случаях неплохо бы иметь *графическое представление инфраструктуры, на которую будет развернуто приложение*. Вот для этого-то и нужны **диаграммы развертывания**, которые иногда называют диаграммами размещения.

Думаю, очевидно, что *такие диаграммы есть смысл строить только для аппаратно-программных систем*, тогда как UML позволяет строить модели любых систем, не обязательно компьютерных.

Какую пользу можно извлечь из диаграмм развертывания? Во-первых, графическое представление ИТ-инфраструктуры может помочь *более рационально распределить компоненты системы по узлам сети*, от чего, как известно, зависит в том числе и производительность системы. Во-вторых, такая диаграмма может помочь *решить множество вспомогательных задач*, связанных, например, с обеспечением безопасности.

Диаграмма развертывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются "трехмерные" обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры. Но приведем пример рисунок 2.24).

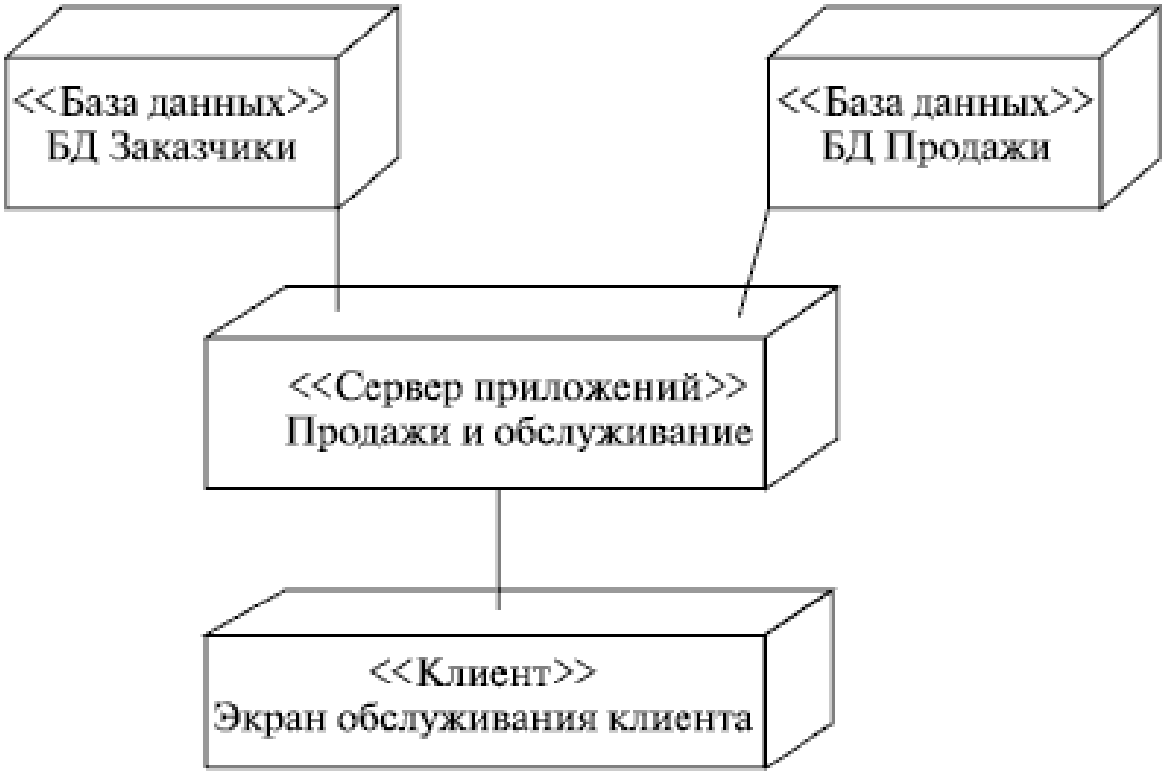


Рисунок 2.24.

Думаем, и без объяснений понятно, что описывает эта диаграмма. А вот *диаграмма развертывания* с большим количеством узлов (рисунок 2.25).

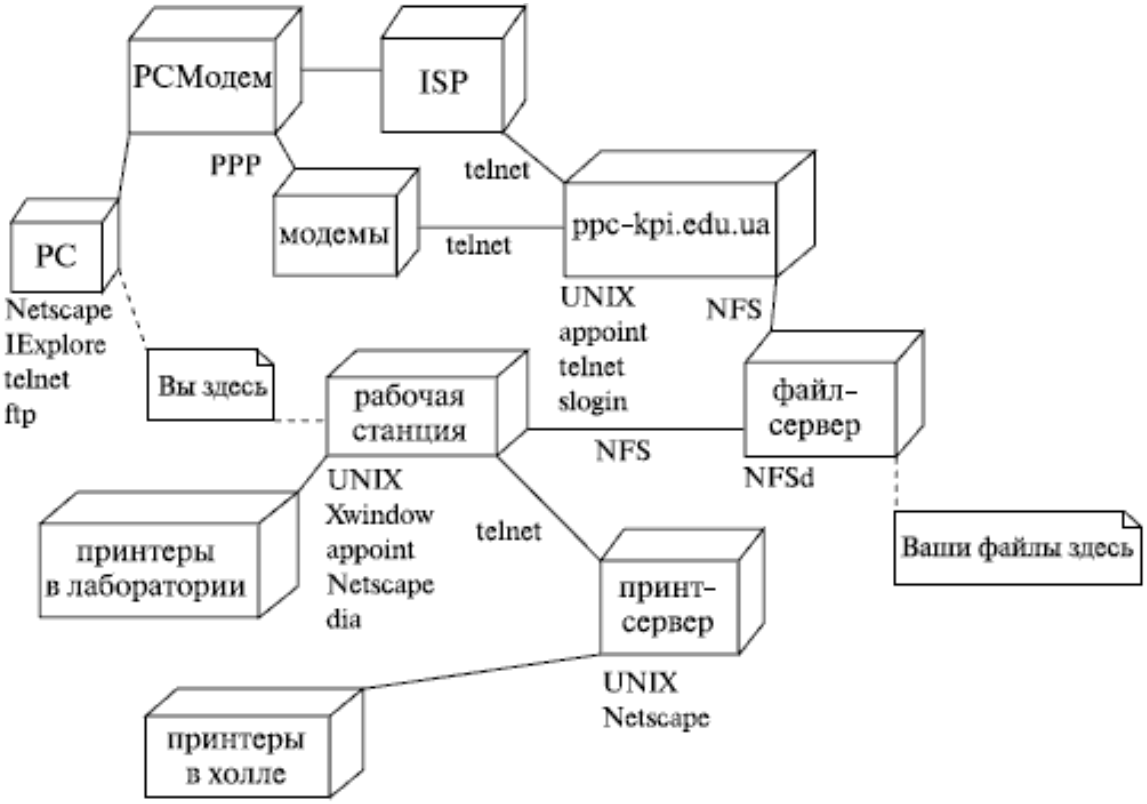


Рисунок 2.25.

И опять все понятно! Это инфраструктура некоего учебного заведения, включающая шлюз, файл-сервер, принт-сервер, принтеры в лабораториях и холле и т. д. Пользователь (вероятно, студент или преподаватель) может получить доступ к этим ресурсам либо со своей домашней машины, либо с рабочих станций, находящихся в лабораториях вуза. Обратите внимание на подписи под линиями, показывающими линии передачи информации, например, видно, что рабочая станция получает доступ к файлам, хранящимся на файл-сервере, посредством NFS. Также хорошая идея - рядом с обозначением узла перечислить программное обеспечение, установленное на данном узле, как это сделано, например, для рабочей станции.

А еще на диаграммах развертывания можно обозначать компоненты системы, т. е. показывать их распределение по аппаратным узлам, но на этом мы пока останавливаться не будем: этих двух примеров уже достаточно, чтобы вы научились распознавать этот вид диаграмм, ведь правда?

ООП и последовательность построения диаграмм

Простые задачи решаются с помощью *UML* без особого труда. А вот более сложные системы, прочитав только этот материал, возможно, так же адекватно смоделировать не удастся. Естественно, *читать об UML недостаточно - надо им пользоваться*! Может быть, даже сразу вы чего-то и не поймете, но *по* мере увеличения опыта использования *UML* вы

все лучше начнете понимать его конструкции. Так же как и другие языки, *UML* требует особого способа мышления, умения рассматривать систему с разных сторон и точек зрения.

Можно дать множество рекомендаций относительно того, какие же именно диаграммы строить и как, но мы будем краткими. Прежде всего, вы должны ответить для себя на такие вопросы:

1. Какие именно виды диаграмм лучше всего отражают архитектуру системы и возможные технические риски, связанные с проектом?
2. Какие из диаграмм удобнее всего превратить в инструмент контроля над процессом (и прогрессом) разработки системы?

И еще одно - *никогда не выбрасывайте даже "забракованные" диаграммы*: они могут в дальнейшем оказаться полезными при анализе направления вашей мысли, поиске ошибок проектирования, да и просто для экспериментов *по* незначительному изменению системы.

Диаграммы, как уже говорилось выше, можно и нужно строить в некоторой логической последовательности. Но как выработать эту последовательность, если у вас нет опыта моделирования? Как научиться этому? Вот несколько простых приемов, которые помогут вам (или вашей команде) выработать свой стиль проектирования.

В *UML*-проектировании, как и при создании любых других моделей, важно уметь **абстрагироваться** от несущественных свойств системы. В этом плане очень полезными могут оказаться *коллективные упражнения на выявление и анализ прецедентов*. Они помогут отработать навыки выявления четких абстракций.

Неплохой способ начать - моделирование базовых абстракций или поведения одной из уже имеющихся у вас систем.

Стройте *модели предметной области задачи в виде диаграммы классов*! Это хороший способ понять, как визуализировать *множества* взаимосвязанных абстракций. Таким же образом стройте модели статической части задач.

Моделируйте *динамическую часть задачи с помощью простых диаграмм последовательностей и кооперации*. Хорошо начать с модели взаимодействия пользователя с системой - так вы сможете легко выделить наиболее важные прецеденты.

Не забываем, что мы говорим, прежде всего, именно об объектноориентированных системах. Поэтому, подытоживая все сказанное ранее, можно предложить такую последовательность построения диаграмм:

- диаграмма прецедентов,
- диаграмма классов,
- диаграмма объектов,
- диаграмма последовательностей,

- диаграмма кооперации,
- диаграмма состояний,
- диаграмма активности,
- диаграмма развертывания.

Конечно, это не единственная возможная последовательность. Возможно, вам будет удобнее начать с *диаграммы классов*. А может, вам не нужны *диаграммы объектов*, а диаграммы последовательностей вы предпочитаете диаграммам кооперации. Это лишь один из путей, постепенно вы выработаете свой персональный стиль проектирования и свою последовательность!

Несколько советов относительно использования *UML*

Хорошее и полезное упражнение - строить модели классов и отношений между ними для уже написанного вами кода на C++ или *Java*.

Применяйте *UML* для того, чтобы прояснить неявные детали реализации существующей системы или использованные в ней "хитрые *механизмы* программирования".

Стройте *UML*-модели, прежде чем начать новый проект. Только когда будете абсолютно удовлетворены полученным результатом, начинайте использовать их как основу для кодирования.

Обратите особое внимание на средства *UML* для моделирования компонентов, параллельности, распределенности, паттернов проектирования. Большинство из этих вопросов мы рассмотрим далее.

UML содержит некоторые средства расширения. Подумайте, как можно приспособить язык к *предметной области* вашей задачи. И не слишком увлекайтесь обилием средств *UML*: если вы в каждой диаграмме будете использовать абсолютно все средства *UML*, прочесть созданную вами модель смогут лишь самые опытные пользователи.

Кроме прочего, важным моментом здесь является выбор пакета *UML*-моделирования (CASE-средства), что тоже может повлиять на ваш индивидуальный стиль проектирования. Более подробно мы поговорим об этом далее, пока же отметим, что все диаграммы, , построены с помощью TAU G2 от Telelogic.

Выводы

1. Диаграммы разных видов позволяют взглянуть на систему с разных точек зрения.
2. UML содержит диаграммы трех типов - для моделирования статической структуры, поведенческих аспектов и подробностей реализации приложения.
3. Недостаточно читать об UML - им надо пользоваться!

Контрольные вопросы

1. Почему нужно строить разные диаграммы при моделировании системы?

2. Какие диаграммы соответствуют статическому представлению о системе?

3. Вы разрабатываете компьютерную программу для игры в шахматы. Какая диаграмма UML была бы полезной в этом случае? Почему?

4. Составьте список вопросов потенциальному пользователю такой программы. Объясните, почему вы хотели бы задать именно их.

Занятие 3. Диаграмма классов: крупным планом

Как класс изображается на диаграмме UML?

Архитектор программного обеспечения в первую *очередь* обращает внимание на объекты *предметной области*. Программист же концентрируется на поведении этих объектов, пользуясь **классами**, к которым они принадлежат. Вот поэтому-то *диаграмма* классов и является одной из важнейших диаграмм *UML*. Она используется для документирования программных систем, и основным ее компонентом является **класс**. Что такое *класс*, мы уже говорили ранее, когда знакомились с видами диаграмм *UML*. Ранее мы рассматривали назначение *диаграммы классов*, знакомились с примерами готовых диаграмм, но не вникали в тонкости обозначений, используемых на диаграмме. В тех примерах все казалось нам очень понятным и логичным. Тем не менее, некоторые нюансы все же следует рассмотреть, и как раз этим мы сейчас и займемся.

Класс на диаграмме изображается в виде прямоугольника, разделенного горизонтальными линиями на три части. В первой части указывается название класса. Как правило, *имя класса* состоит из одного, *максимум* двух слов. Вторая часть содержит перечень атрибутов класса, которые характеризуют тот или иной *объект* этого класса в модели *предметной области*. Третья часть содержит перечень операций, отражающих его поведение в модели *предметной области* (рисунок 3.1). Все очень просто, не так ли?

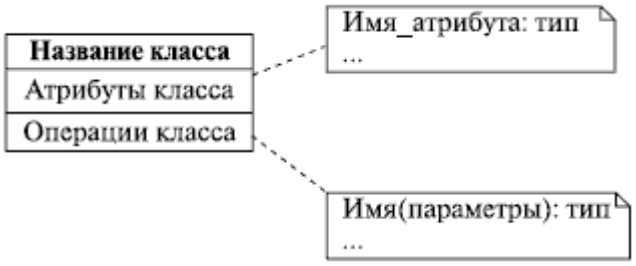


Рисунок 3.1.

А что внутри?

Мы узнали, как *класс* изображается и выглядит "снаружи". А что же внутри объектов класса? Пользователю об этом знать необязательно, более того, абсолютно не нужно. Для человека, использующего его, *объект* выступает в роли черного ящика. Скрывая от пользователя внутреннее устройство объекта, мы обеспечиваем его надежную работу. Сейчас мы рассмотрим, как убрать из поля зрения пользователя то, что ему знать не нужно.

Читателя может слегка смутить *слово* "*пользователь*", которым мы злоупотребляли в предыдущем абзаце. Зачем вообще пользователю какие-то объекты и классы? Внесем *ясность*. Программист, использующий в своей программе созданные кем-то компоненты, как раз и выступает в роли такого пользователя. Зачем ему знать что внутри - он знает, какие

атрибуты надо модифицировать и какие *операции* использовать, чтобы заставить *объект* работать именно так, как ему нужно! Более того, а многие ли из нас знают, как именно устроен и по каким принципам работает, например, *телевизор* - *объект* класса "Бытовой прибор"?

Сокрытие от пользователя внутреннего устройства объектов называется *инкапсуляцией*. Если говорить более "научным" языком, то *инкапсуляция* - это защита отдельных элементов объекта, не затрагивающих существенных характеристик его как целого. *Инкапсуляция* нужна не только для того, чтобы создать иллюзию простоты объекта для пользователя (по словам Г. Буча). Но вернемся к примеру с телевизором. Нам этот прибор кажется очень простым только потому, что при работе с ним мы используем простой и понятный *интерфейс* - пульт дистанционного управления. Мы знаем: для того чтобы увеличить громкость звука, надо нажать вот эту кнопку, а чтобы переключить канал - вот эту. Как *телевизор* устроен внутри, мы не знаем. Более того - в отсутствие пульта ДУ такое *знание* было бы неудобным для нас и весьма опасным для самого телевизора, вздумай мы увеличить громкость с помощью паяльника. Поэтому-то пульт ДУ и защищает от нас "внутренности" телевизора! Вот так *инкапсуляция* реализуется в реальном мире.

В программировании *инкапсуляция* обеспечивается немного по-другому - с помощью т. н. *модификаторов видимости*. С их помощью можно ограничить *доступ* к атрибутам и операциям объекта со стороны других объектов. Звучит это немного пугающе, но на самом деле все просто. Если *атрибут* или операция описаны с модификатором private, то *доступ* к ним можно получить только из *операции*, определенной в том же классе. Если же *атрибут* или операция описаны с модификатором видимости public, то к ним можно получить *доступ* из любой части программы. Модификатор protected разрешает *доступ* только из операций этого же класса и классов, создаваемых на его основе. В языках программирования могут встречаться *модификаторы видимости*, ограничивающие *доступ* на более высоком уровне, например, к классам или их группам, однако смысл инкапсуляции от этого не изменяется. В *UML* атрибуты и *операции* с модификаторами доступа обозначаются специальными символами слева от их имен:

Символ	Значение
+	public - открытый доступ
-	private - только из операций того же класса
#	protected - только из операций этого же класса и классов, создаваемых на его основе

Рассмотренный ранее пример с телевизором средствами *UML* (конечно же, это очень высокоуровневая *абстракция*) можно изобразить так (рисунок 3.2):

Рисунок 3.2.

Не правда ли, все понятно и предельно просто? Зачем, например, пользователю знать числовые значения частот каналов? Он знает, что достаточно запустить процедуру автоматического поиска каналов и *телевизор* все сделает за него. Вот вам и *инкапсуляция* - оказывается, она повсюду вокруг нас. Оглянитесь и подумайте, сколько вещей вокруг имеют скрытые свойства и выполняют скрытые *операции*. Испугались? Вот то-то же!

Как использовать объекты класса?

Итак, мы рассмотрели инкапсуляцию - одно из средств защиты объектов. Все вроде бы понятно, но как же именно работать с объектом?

Если уж говорить о защите объекта, то чтобы она действительно была эффективной, надо позаботиться о некоем стандартном и безопасном, не зависящим от языка программирования способе доступа к объекту. К тому же такой стандартный способ доступа должен быть простым и с точки зрения использования, и с точки зрения реализации. Вспомните пример с телевизором. Нажимая кнопки на пульте, мы ожидаем, что *телевизор* откликнется на это действие каким-то определенным образом - именно так, как мы ожидаем, а не иначе. То есть, с одной стороны, пульт ДУ является средством доступа к скрытым операциям, выполняемым телевизором, а с другой стороны - пульт обеспечивает нужное для нас поведение телевизора. В данном примере именно пульт является таким стандартным средством доступа к телевизору. Можно даже сказать, средством доступа, не зависящим от конкретной модели телевизора - вспомните об универсальных пультах и о том, как отключаете звук надоедливой рекламы на экране в вагоне поезда, используя КПК!

В том же примере с телевизором у нас впервые промелькнуло *слово интерфейс*. И не случайно промелькнуло: именно так называют тот самый стандартный способ доступа к объекту. Более строго, *интерфейс* - это логическая *группа* открытых (public) операций объекта. Один и тот же *объект* может иметь несколько интерфейсов. У телевизора, например, их два

- пульт ДУ и кнопки на корпусе. А может и больше - вспомните о возможности управлять бытовой техникой с помощью КПК или универсального пульта ДУ.

Кстати, посмотрите внимательнее на пульт ДУ или на экран программы удаленного контроля. Что вы видите - кнопки? Или кнопки, сгруппированные по функциональному признаку? Да, именно так: кнопки, переключающие каналы, расположены отдельно, рядом - *группа* кнопок, отвечающих за регулировку громкости звука, рядом - *группа* программируемых кнопок, и т. д. В принципе, можно сказать, что пульт реализует не один, а несколько интерфейсов - по числу функциональных групп кнопок. Впрочем, это уже *формализм*: мы просто хотели проиллюстрировать слова "логическая *группа*" в определении интерфейса.

Однако *интерфейс* - это не только и не столько *группа* операций объекта. *Интерфейс* отражает внешние проявления объекта, показывает, каким образом осуществляется взаимодействие с ним, скрывая остальные детали, не имеющие отношения к процессу взаимодействия.

Интерфейс всегда реализуется некоторым классом, который в таком случае называют классом, *поддерживающим интерфейс*. Как мы уже говорили ранее, один и тот же *объект* может иметь несколько интерфейсов. Это означает, что *класс* этого объекта реализует все *операции* этих интерфейсов. К данному моменту в голове читателя может созреть вопрос: "Мы же, вроде бы, говорили о классах и объектах, а теперь вдруг перешли на интерфейсы. Да и вообще, используются ли они в практике программирования или являются просто изящной теоретической конструкцией?". Ответ на этот вопрос прост: многие из существующих технологий программирования (например, *COM, CORBA, Java Beans*) не только активно используют *механизм интерфейсов*, но и, по сути, полностью основаны на нем.

Что ж, наверное, пришло время поговорить о том, как *интерфейс* изображается на диаграммах. Изображаться он может несколькими способами. Первый и самый простой из них - это *класс* со стереотипом <<interface>> (рисунок 3.3).

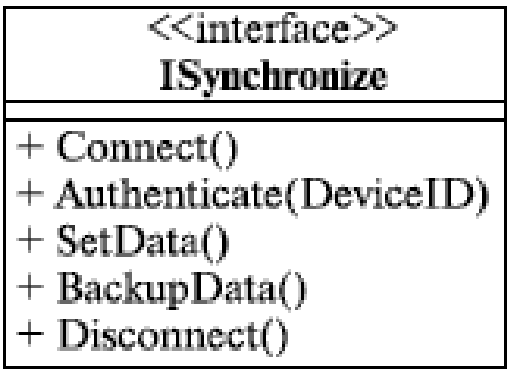


Рисунок 3.3.

Этот способ хорош, если нужно показать, какие именно *операции* предоставляет *интерфейс*. Если же такие подробности в данный момент не важны, предоставляемый *интерфейс* изображают в виде кружочка или, как говорят, "леденца" (lollipop) (рисунок 3.4).

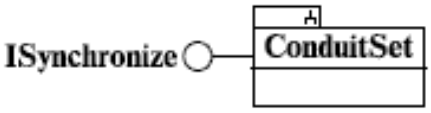


Рисунок 3.4.

Обратите внимание на маленький значок на закладке папки ConduitSet. Это обозначение подсистемы, мы могли бы не рисовать его, а просто использовать стереотип <<subsystem>>. Впрочем, об этом мы еще поговорим.

И наконец, еще один способ изображения интерфейса. Он не является альтернативой описанным ранее способам, а используется для изображения интерфейсов, **требующихся** объекту для выполнения его работы. Обозначается он очень простым и логичным символом. Впрочем, судите сами (рисунок 3.5).

Рисунок 3.5.

Наблюдательный читатель уже, наверное, заметил, как логически совмещаются символы предоставляемого и требуемого интерфейсов.

Действительно, на диаграммах довольно часто можно увидеть такую картинку (рисунок 3.6):

Рисунок 3.6.

Да, кстати, вы заметили, что названия интерфейсов начинаются с буквы I? Эта традиция пошла из языка *Java*, и, как показывает практика, она весьма облегчает жизнь, если нужно, например, быстро разобраться в сложной диаграмме, составленной другим человеком.

Всегда ли нужно создавать новые классы?

Начнем с вопроса, казалось бы, не имеющего никакого отношения к рассматриваемому вопросу, а именно - всегда ли нужно создавать новый *класс* для каждой новой задачи? *Правильный ответ*, конечно же, "нет". Это было бы странно и неэффективно. "Фишка" состоит в том, что мы можем использовать уже существующие классы, адаптируя их функциональность для выполнения новых задач. Таким образом появляется возможность не со-

здавать систему классов с нуля, а задействовать уже имеющиеся решения, которые были созданы ранее, при работе над предыдущими проектами. Впрочем, наше *высказывание* о странности и неэффективности создания новых классов не является истиной в последней инстанции. Могут быть ситуации, когда существующие классы по каким-либо причинам не устраивают архитектора, и тогда требуется создать новый *класс*. Следует, однако, избегать ситуаций, когда созданный *класс* (а точнее, его набор операций и атрибутов) практически повторяет существующий, лишь незначительно отличаясь от него. Все-таки лучше не изобретать велосипед и стараться создавать классы на основе уже существующих, и только если подходящих классов не нашлось - создавать свои, которые, в свою *очередь*, могут (и должны!) служить основой для других классов. Мы уже не говорим о том, что создание классов предполагает значительный объем усилий по кодированию и тестированию. В общем случае, сказанное выше можно проиллюстрировать такой диаграммой (рисунок 3.7):

Рисунок 3.7.

В *дополнение* можно назвать несколько причин, почему стоит использовать уже существующие классы:

Во-первых, идя этим путем, мы пользуемся плодами ранее принятых решений. Действительно, если когда-то мы уже решили некоторую проблему, зачем начинать все "с нуля", повторяя уже однажды проделанные действия?

Во-вторых, таким образом мы делаем решение мобильным и расширяемым. Используя уже существующие классы и создавая на их основе новые, мы можем развивать решение практически неограниченно, добавляя лишь необходимые нам в данный момент детали - атрибуты и *операции*.

В-третьих, существующие классы, как правило, хорошо отлажены и показали себя в работе. Разработчику не надо тратить время на *кодирование*, отладку, тестирование и т. д., - мы работаем с хорошо отлаженным и проверенным временем кодом, который зарекомендовал себя в других проектах и в котором уже выявлено и исправлено большинство ошибок.

А теперь внимание - мы много говорили о том, что нужно создавать классы на основе уже существующих, но так и не сказали ни слова о том, как это сделать. Пришло время внести *ясность* в этот вопрос. Тем самым мы подбираемся к понятию **обобщения** или **генерализации**, которое играет очень важную роль в *ООП*, являясь одним из его базовых принципов. **Обобщение** - это *отношение* между более общей сущностью, называемой *суперклассом*, и ее конкретным воплощением, называемым *подклассом*. Иногда *обобщение* называют отношениями типа "является", имея в виду, что одни сущности (например, круг, квадрат, треугольник) являются воплощением более общей сущности (например, класса "*геометрическая фигура*"). При этом все атрибуты и *операции* суперкласса независимо от *модификаторов видимости* входят в состав подкласса.

Обобщение (или, как часто говорят, *наследование*) на диаграммах обозначается очень просто - незакрашенной треугольной стрелкой, направленной на *суперкласс* (рисунок 3.8).

Для того чтобы научиться эффективно моделировать *наследование*, обратимся к классикам, а именно к Г. Бучу. Он советует проводить эту процедуру в такой последовательности:

1. Найдите атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности. Это позволит избежать ненужного дублирования структуры и функциональности объектов.
2. Вынесите эти элементы в некоторый общий суперкласс, а если такого не существует, то создайте новый класс.
3. Отметьте в модели, что подклассы наследуются от суперкласса, установив между ними отношение обобщения.

Рисунок 3.8.

А вот и пример применения этого подхода (рисунок 3.9):

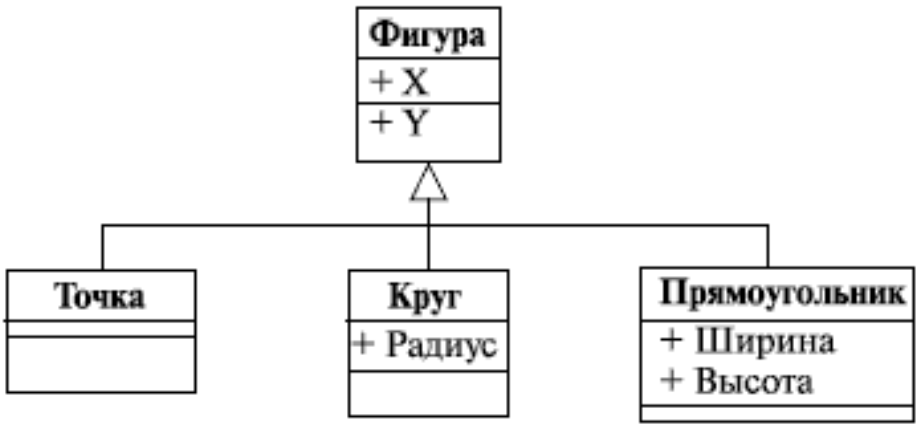


Рисунок 3.9.

На первый взгляд, кажется странным, что *класс* "точка" не имеет никаких атрибутов, а круг имеет только *радиус*. С прямоугольником, вроде бы, все понятно - ширина и *высота*, но вот только где он расположен в пространстве, этот *прямоугольник*? Давайте попробуем следовать советам Буча. Итак, положение всех трех фигур можно однозначно определить с помощью пары чисел. Для точки - это вообще единственные ее характеристики, для круга и прямоугольника - их центры (под центром прямоугольника мы понимаем точку пересечения его диагоналей). Вот они, общие атрибуты! Таким образом, мы создали *суперкласс* "Фигура", имеющий два атрибута - *координаты* центра. Все остальные классы на этой диаграмме связаны с классом "Фигура" отношением обобщения, т. е. в них нужно доопределить только "недостающие" атрибуты - *радиус*, ширину и высоту. Атрибуты, описывающие *координаты* центра, эти классы имеют изначально как потомки класса "Фигура" - они их наследуют. Заметим, что *операции* классов мы тут не рассматриваем: понятно, что с ними была бы та же история.

Так, с наследованием вроде бы разобрались. Пришло время для маленькой провокации с нашей стороны. Классы-потомки ведь наследуют атрибуты и *операции* суперкласса? Таким образом, они могут наследовать и их интерфейсы - то есть объекты абсолютно разной природы могут иметь один и тот же *интерфейс*! Так как же тогда определить, какого же все-таки класса *объект*? Да и нужно ли это вообще?

Действительно, объекты разной природы (или говоря проще, разных классов) могут поддерживать один и тот же *интерфейс* именно так, как того ожидает *пользователь*. Примером тому может служить рассмотренная выше *диаграмма* с геометрическими фигурами. Все рассмотренные фигуры имеют, например, операцию рисования на экране. С точки зрения пользователя в каждом случае это одно и то же действие. Однако реализованы эти *операции* по-разному - ведь процедура изображения прямоугольника сильно отличается от подобной процедуры для круга. Но для пользователя это неважно: ведь *сигнатура*-то одна и та же! А возможно это благодаря еще одному из основных принципов *ООП* - **полиморфизму.**

Как мы только что упомянули, работа механизма полиморфизма основана на совпадении сигнатуры метода, объявленного в интерфейсе, и сигнатуры самого метода. Методы внутри классов-потомков могут быть (и наверняка будут!) переопределены, их реализации будут различными, а сигнатуры останутся неизменными. Таким образом (и в этом легко ощутить мощь *ООП*), выполняя одни и те же *операции*, разные объекты могут вести себя по-разному.

Полиморфизм является основой для реализации *механизма интерфейсов* в языках программирования. Вот, кстати, и ответ на вопрос, какого класса *объект*: как только *пользователь*обращается к некоторой *операции* через *интерфейс*, определяется фактический *класс* объекта и вызывается соответствующая *операция класса*. Примеры полиморфизма можно увидеть в самых обыденных вещах, которыми мы пользуемся в повседневной жизни. Оглянитесь вокруг - мир построен по *ООП, Матрица* работает! Например, всем привычная кредитная карточка, является интерфейсом для доступа к банковскому счету через банкомат (и не только), одинаково работает в любой стране, вот только ведет себя чуть-чуть по-разному, т. к. банкомат выдает деньги в местной валюте. Согласны, пример не очень корректный, но зато очень наглядный! Думаем, понаблюдав за окружающим миром, читатель сам сможет привести массу примеров полиморфизма.

Инкапсуляция, наследование и *полиморфизм*, с которыми мы только что познакомились, являются теми самыми тремя китами, на которых держится *ООП*. Если вы поняли суть этих базовых принципов и осознали их истинную мощь, вы прошли большую часть пути, ведущего к полному овладению *ООП* как наиболее адекватной методикой описания (так и тянет сказать "проектирования") окружающего нас мира.

Отношения между классами

Ни один из объектов окружающего нас мира не существует сам по себе. Птицы летают потому, что есть воздух, на который опираются их крылья. Каждый из нас связан с массой других людей разнообразными родственными, профессиональными и другими связями, предполагающими различные типы отношений. Точно так же и классы связаны между собой. И чтобы в полной мере овладеть *ООП*, нам необходимо понять суть этих отношений и научиться их идентифицировать.

Мы сказали, что объекты находятся в определенных отношениях друг с другом. Один из типов таких отношений - это **зависимость**. Думаем, суть такого отношения понятна уже из его названия - зависимость возникает тогда, когда реализация класса одного объекта зависит от спецификации операций класса другого объекта. И если изменится спецификация операций этого класса, нам неминуемо придется вносить изменения и в зависимый *класс*. Приведем простой пример, опять-таки взятый из нашей повседневности. Иногда к нам в руки

попадают видеофайлы, воспроизвести которые "с лету" не удается. Почему? Правильно, потому что на компьютере не установлены соответствующие кодеки. То есть операция "Воспроизведение", реализуемая программой-медиаплеером, зависит от *операции* "Декомпрессия", реализуемой кодеком. Если спецификация *операции* "Декомпрессия" изменится, придется менять код медиаплеера, иначе он просто не сможет работать с каким-то кодеком и, в лучшем случае, завершит свою работу с ошибкой. А вот так зависимость между классами изображается в *UML* (рисунок 3.10):

Рисунок 3.10.

Стоит отметить, что зависимости на диаграммах изображают далеко не всегда, а только в тех случаях, когда их *отображение* является важным для понимания модели. Часто зависимости лишь подразумеваются, т. к. логически следуют из природы классов.

Другой вид отношений между объектами - это **ассоциация**. Это просто *связь* между объектами, по которой можно между ними перемещаться. *Ассоциация* может иметь имя, показывающее природу отношений между объектами, при этом в имени может указываться *направление* чтения связи при помощи треугольного маркера. Однонаправленная *ассоциация* может изображаться стрелкой. Проиллюстрируем сказанное примерами (рисунок 3.11):

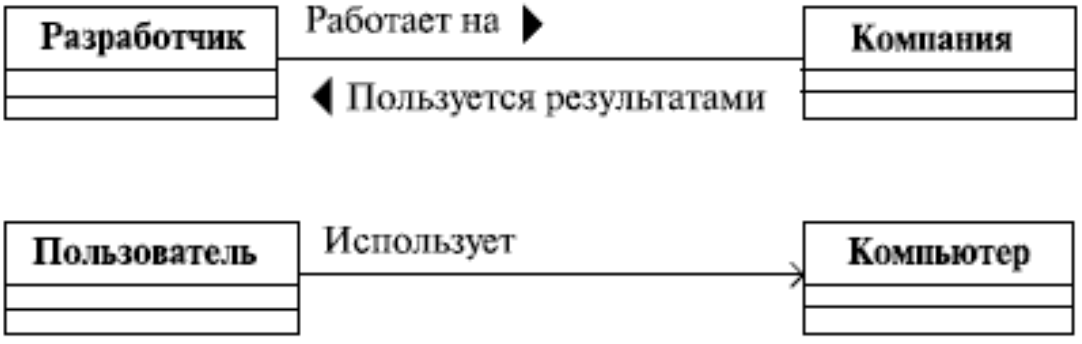


Рисунок 3.11.

Кроме направления ассоциации, мы можем указать на диаграмме *роли*, которые каждый *класс* играет в данном отношении, и *кратность*, то есть количество объектов, связанных отношением (рисунок 3.12):

Рисунок 3.12.

И насчет ролей, и насчет кратности на этой диаграмме все понятно - человек может вообще не работать, работать в одной или более компаниях, а вот компании в любом случае нужен хотя бы один сотрудник. Кстати, о кратности. *Ассоциация* может объединять три и более класса. В этом случае она называется **n-арной** и изображается ромбом на пересечении линий, как показано на этой диаграмме, позаимствованной нами из Zicom Mentor (рисунок 3.13):

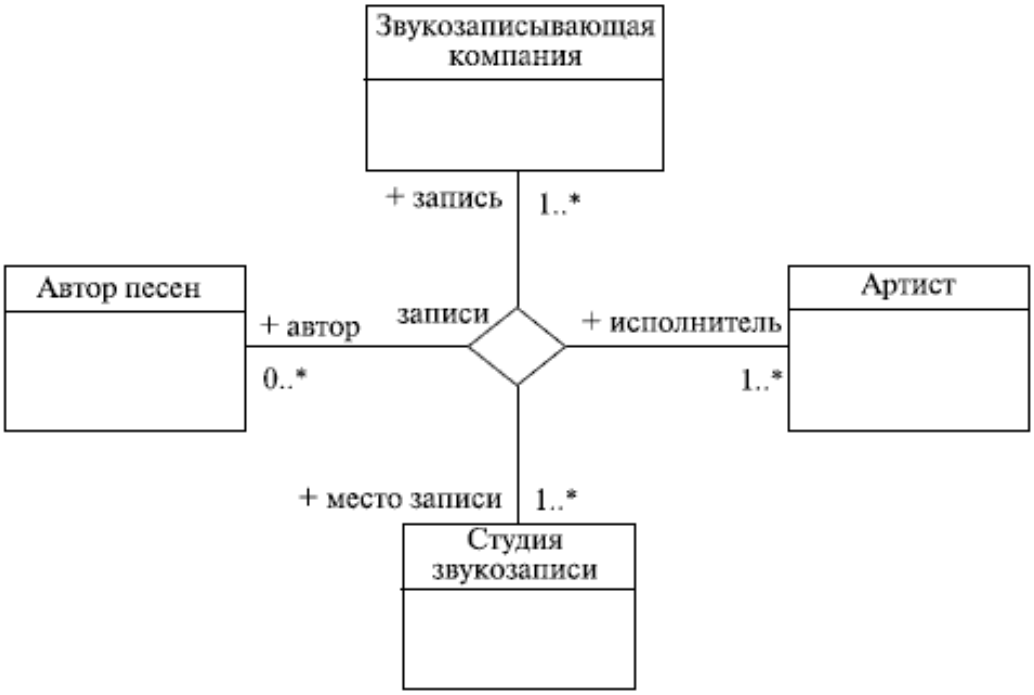


Рисунок 3.13.

Ранее мы говорили, что *ассоциация* - это "просто *связь*" между объектами. На самом деле, в реальности связи бывают "просто связями" крайне редко. Обычно при ближайшем рассмотрении под ассоциацией понимается более сложное *отношение* между классами,

например, *связь* типа "часть-целое". Такой вид ассоциации называется **ассоциацией с агрегированием**. В этом случае один *класс* имеет более высокий статус (целое) и состоит из низших по статусу классов (частей). При этом выделяют простое и композитное *агрегирование* и говорят о собственно **агрегации** и **композиции**. Простая *агрегация* предполагает, что части, отделенные от целого, могут продолжать свое существование независимо от него. Под композитным же агрегированием понимается ситуация, когда целое владеет своими частями и их время жизни соответствует времени жизни целого, т. е. независимо от целого части существовать не могут. Примеры этих видов ассоциаций и их обозначений в *UML* можно увидеть на следующей диаграмме (рисунок 3.14).

Рисунок 3.14.

Примеры, как нам кажется, очень простые и понятные. *Винчестер* можно вынуть из компьютера и установить в новый *компьютер* или в USB-карман, т. е. существование жесткого диска с разборкой системного блока не заканчивается. А вот кнопки без окон обычно существовать не могут - с закрытием окна кнопки также исчезают.

И, наконец, еще одна важная вещь, касающаяся ассоциации. В отношении между двумя классами сама *ассоциация* тоже может иметь свойства и, следовательно, тоже может быть представлена в виде класса. Пример прост (рисунок 3.15).

Рисунок 3.15.

Действительно, перед началом трудовых отношений работник и работодатель подписывают между собой контракт, который имеет такие атрибуты, как, например, описание *работ*, сроки их выполнения, порядок оплаты и т. д.

А вот более сложный, но, опять-таки, взятый из реальной жизни пример моделирования отношений между классами, позаимствованный нами из Zicom Mentor (рисунок 3.16):

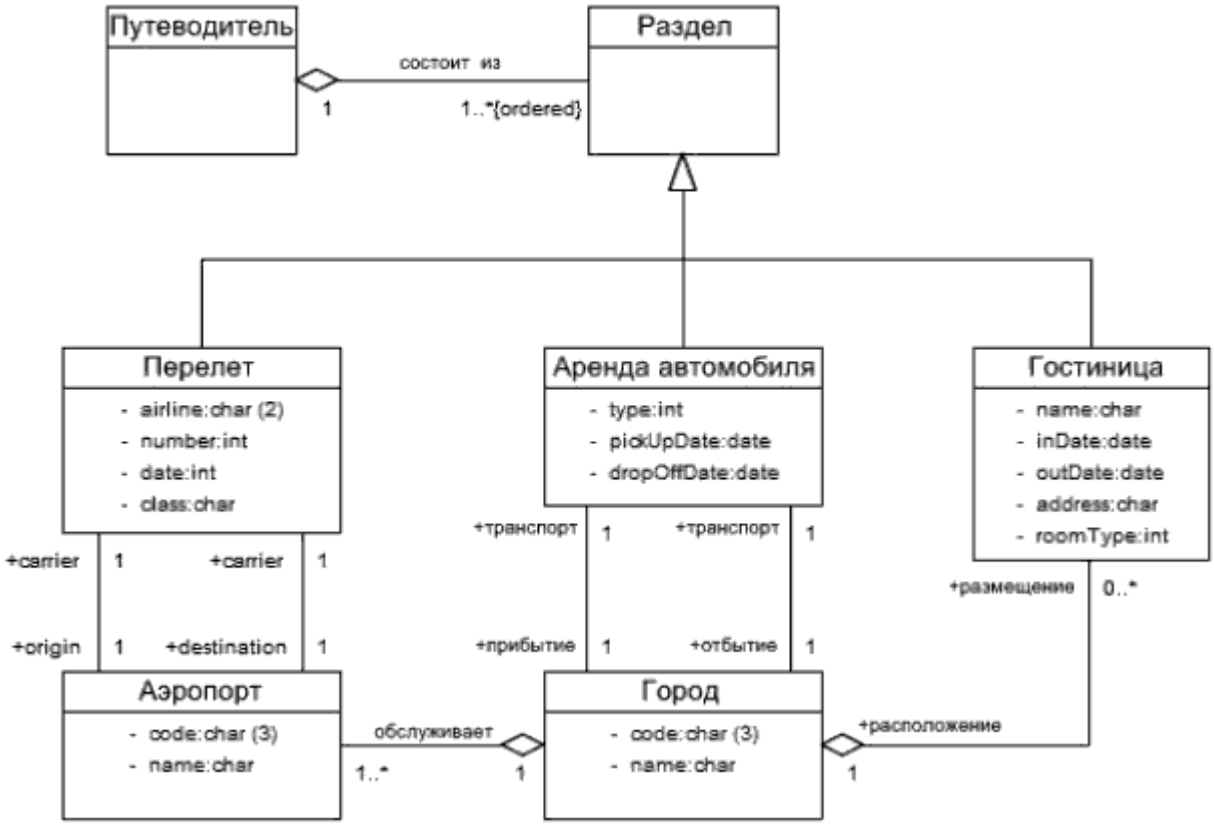


Рисунок 3.16.

И наконец, *доказательство* того, что *UML* можно использовать для чего угодно, в том числе и для записи сказок: *диаграмма*, описывающая предметную область сказки о Курочке Рябе и взятая с сайта конкурса шуток на UML (http://www.umljokes.com/) (рисунок3.17):

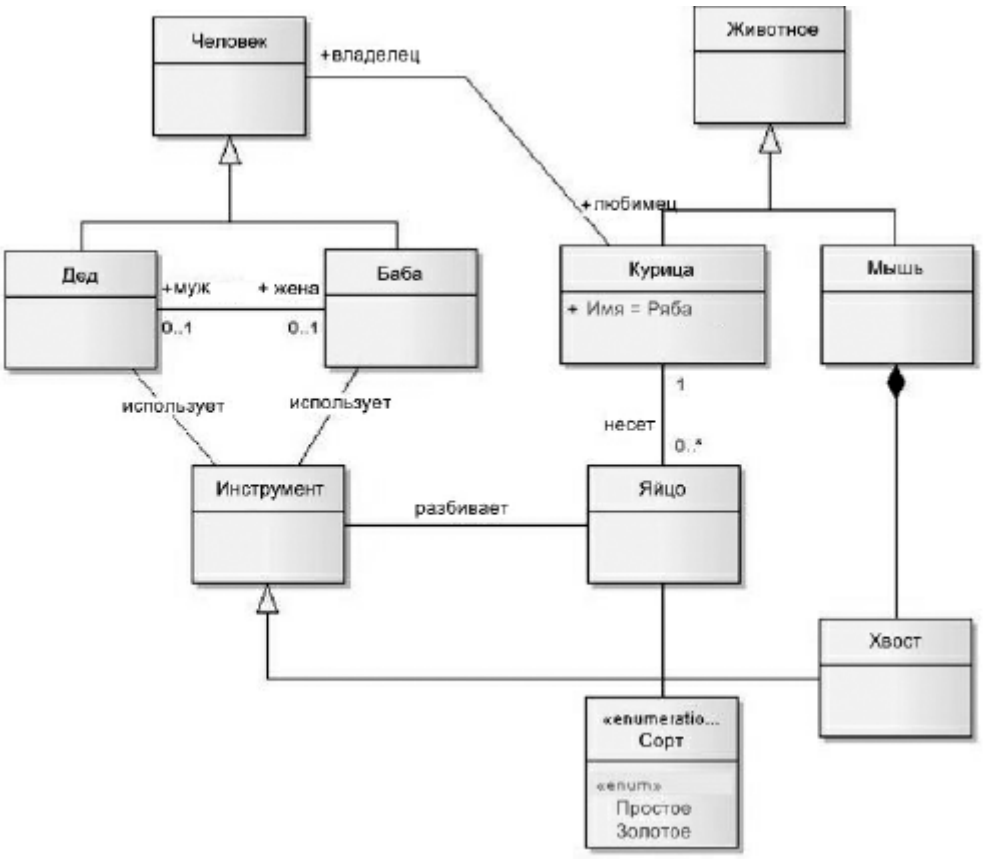


Рисунок 3.17.

Узнаете рассказ, знакомый с детства?

Выводы

1. Инкапсуляция защищает внутреннее устройство объекта и реализуется путем ограничения доступа к атрибутам и операциям класса из других частей программы.
2. Обобщение позволяет повторно использовать уже существующие решения, создавая новые классы путем наследования от имеющихся классов.
3. Полиморфизм позволяет работать с группой разнородных объектов одинаковым образом, не задумываясь о различиях в реализации.
4. Инкапсуляция, наследование и полиморфизм - три кита, на которых держится ООП.
5. В любой системе между объектами существуют отношения разных типов.
6. Отношение зависимости означает, что реализация одного класса зависит от спецификации операций другого класса.
7. Ассоциация выражает отношение между несколькими равноправными объектами и может иметь направление, роли и кратность, а также изображаться в виде класса ассоциации.

8. Композиция и агрегация используются, если между объектами существуют отношения типа "часть-целое", причем композиция предполагает, что части не могут существовать отдельно от целого.

Контрольные вопросы

1. Какие три принципа лежат в основе ООП?
2. Что такое интерфейс? На каком из базовых принципов ООП основан *механизм интерфейсов*?
3. Что такое n-арная ассоциация?
4. В чем разница между агрегацией и композицией?
5. Что такое класс ассоциации?

Занятие 4. Диаграмма активностей

Как мы уже говорили, *диаграммы активностей* (*Activity Diagrams*) являются представлением алгоритмов неких действий (активностей), выполняющихся в системе. Мы уже знаем, что *нотация UML* предлагает пять представлений системы:

1. Вид системы с точки зрения *прецедентов*.
2. Вид с точки зрения *проектирования*.
3. Вид с точки зрения *процессов*.
4. Вид с точки зрения *развертывания*.
5. Вид с точки зрения *реализации*.

И при этом каждый из перечисленных способов представления системы может содержать последовательности действий, которые могут быть описаны с помощью алгоритмов. Вот здесь-то и выходят на сцену диаграммы деятельностей. Вообще говоря, любой элемент модели, имеющий динамическое поведение, может быть дополнен диаграммой деятельности - именно для уточнения этой самой динамики. Как хорошо подходящий *по* контексту пример следует упомянуть возможность применения диаграмм активности для описания бизнес-процессов, существующих в компании (нотации Grapes-BM, *BPML*/BPMN и др.). Вот уж где самая что ни на есть динамика!

Можно построить несколько диаграмм деятельности для одной и той же системы, причем каждая из них будет фокусироваться на разных аспектах системы, показывать различные действия, выполняющиеся внутри ее. Читатель, конечно же, понял, что, когда мы говорим о *динамике*, мы подразумеваем *поведение* системы в целом или ее частей. Говоря более формально, диаграммы активности, в общем-то, не имеют монополии на описание поведенческих особенностей динамических частей системы. Для этой же цели могут использоваться еще *диаграммы прецедентов*, последовательности, кооперации и состояний. Почему же мы говорим именно о диаграмме активности? Именно на диаграмме деятельности представлены переходы потока управления от одной деятельности к другой. Это, *по* сути, разновидность диаграммы состояний, где все или большая часть состояний являются некоторыми деятельностями, а все или большая часть переходов срабатывают при завершении определенной деятельности и позволяют перейти к выполнению следующей. Как мы уже говорили (повторение - мать учения), *диаграмма* деятельности может быть присоединена к любому элементу модели, имеющему динамическое поведение. Кстати, исходя из вышесказанного, логичнее говорить не "*диаграмма* деятельности", а "*диаграмма* деятельностей" - во множественном числе. А еще мы предполагаем, что читатель понимает смысл понятий "*деятельность*", "переход" и "*объект*". Об объектах как об экземплярах классов мы уже говорили

ранее. Понятия же деятельности (*activity*) как протяженного во времени составного (неатомарного) вычисления (действия, action) и перехода как передачи контроля, надеемся, понятны интуитивно, без дополнительных объяснений.

Диаграммы деятельности позволяют моделировать сложный **жизненный цикл объекта**, с переходами из одного состояния (деятельности) в другое. Но этот вид диаграмм может быть использован и для описания динамики совокупности объектов. Они применимы и для детализации некоторой конкретной *операции*, причем, как мы увидим далее, предоставляют для этого больше возможностей, чем "классическая" *блок-схема*. Диаграммы деятельности описывают переход *от одной деятельности к другой*, в отличие от диаграмм взаимодействия, где акцент делается на переходах потока управления *от объекта к объекту*.

Как говорится, лучше один раз увидеть, чем сто раз услышать. Мы достаточно разрекламировали диаграммы деятельностей. Пора взглянуть на пример (рисунок4.1).

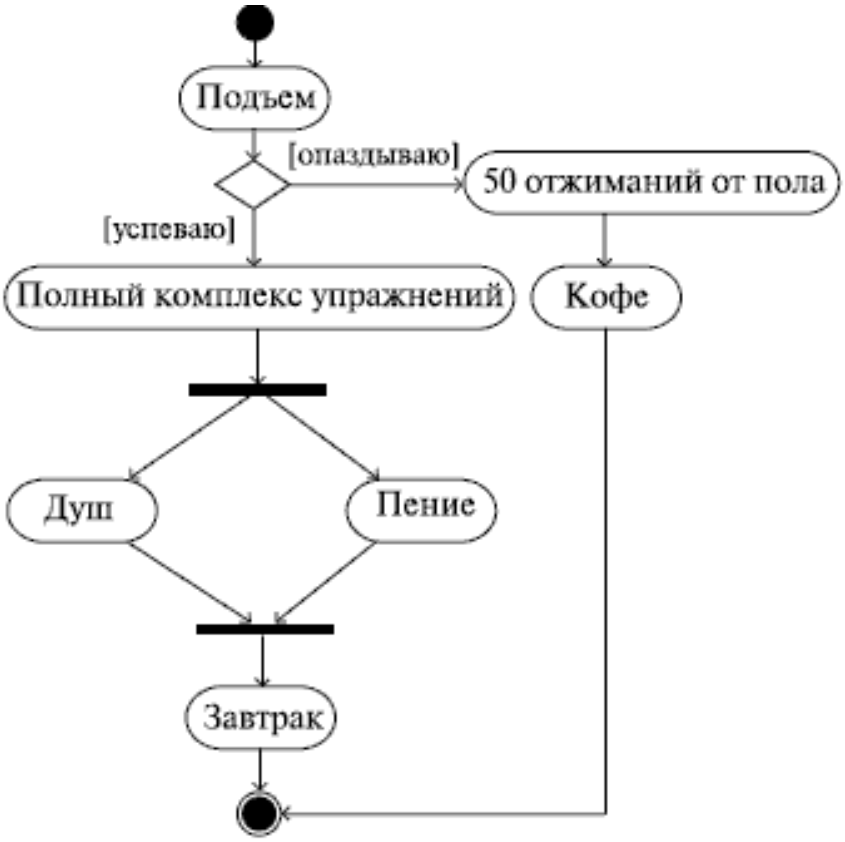


Рисунок 4.1.

Эта *диаграмма* довольно точно описывает ежеутреннюю последовательность действий автора этих строк (до момента ухода на работу). Как видим, все очень просто и понятно. Действия показаны скругленными прямоугольниками, как в блок-схеме, - мы узнаем даже ромбик символа принятия решения с обозначениями условий возле переходов. Да, отличия от блок-схемы не так уж сильны. Более того, эти отличия выглядят как логичное расширение нотации блок-схем. Обратим внимание на то, что начало и конец уже не изображаются одинаковым безликим кружком. Начало теперь закрашено, а конец изображен в виде

символа, напоминающего кошачий глаз (рисунок 4.2) (кстати, это образное название - "кошачий глаз" - уже намертво въелось в жаргон архитекторов и аналитиков).

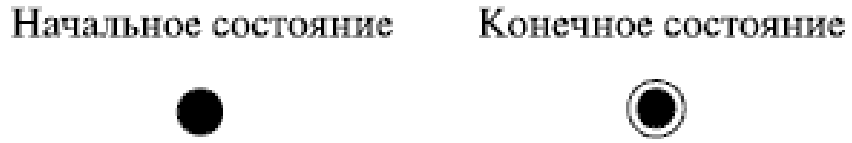


Рисунок 4.2.

Без пояснений понятен также смысл символа, предшествующего принятию душа и пению и следующего за ними - он означает *распараллеливание*, а затем опять слияние воедино (*синхронизацию*) потоков управления, т. е. *операции* "пение" и "душ" выполняются *одновременно*. *Нотация* проста: несколько потоков управления сливаются в один или один *поток*разделяется на несколько. Третьего не дано (рисунок 4.3).

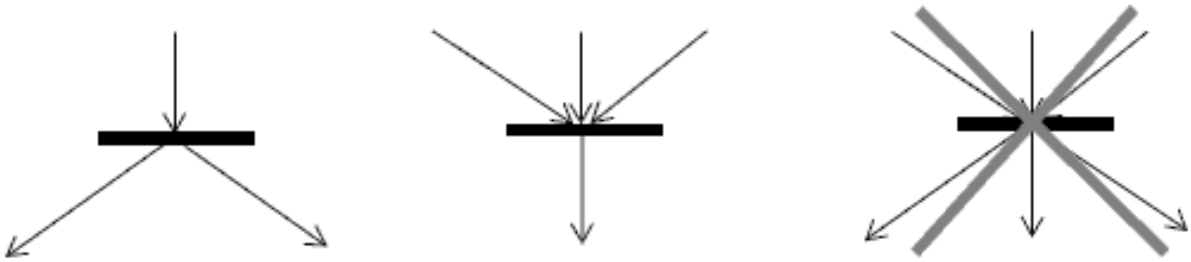


Рисунок 4.3.

Конечно, это не единственные отличия *диаграммы активностей* от блок-схемы. На диаграмме деятельностей можно не только показать параллельно выполняемые действия, но и указать состояния объектов (так же, как и на представлениях конечных автоматов, о которых нам так много говорили в университетах), также есть возможность показывать распределение ролей и т. д. Вот еще пример, подтверждающий, что *диаграмма* активностей - это нечто большее, чем *блок-схема* (рисунок 4.4).

Смысл диаграммы вполне понятен и без дополнительных объяснений. Как вы уже, конечно, догадались, на ней показана работа с веб-приложением, которое решает некую задачу в удаленной базе данных. Привлекает внимание странное расположение активностей на этой диаграмме: они как бы разбросаны *по* трем беговым дорожкам, каждая из которых соответствует поведению одного из трех объектов - клиента, веб-сервера и сервера баз данных. Благодаря этому легко определить, каким из объектов выполняется каждая из активностей, и неожиданно приходит понимание того, что "странность" этой диаграммы, оказывается, очень упрощает ее восприятие.

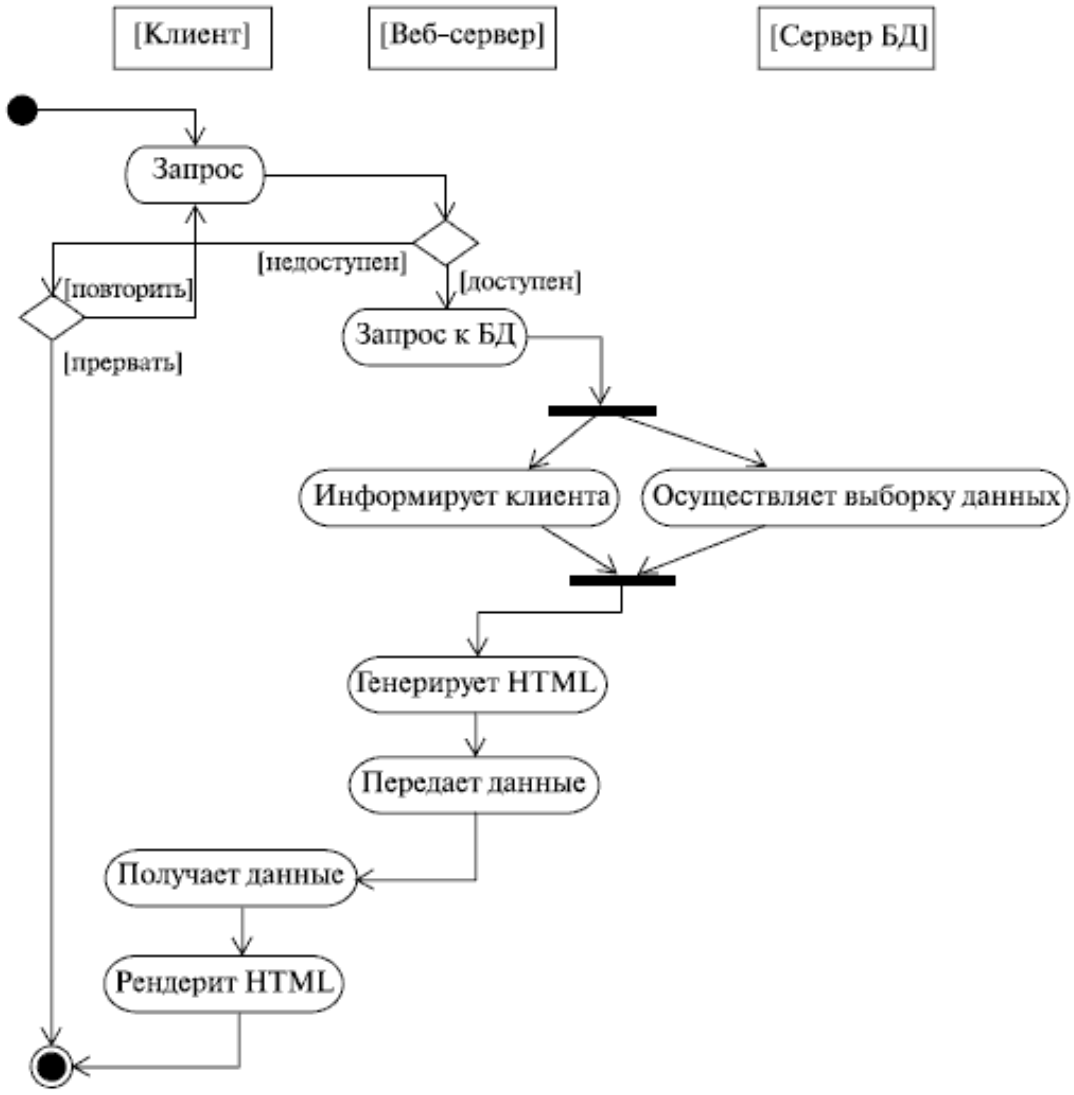


Рисунок4.4.

Аналогия с дорожками действительно очень удачна. Именно таково официальное название элемента нотации *UML*, позволяющего указать распределение ролей на диаграмме активностей. Только дорожки это не беговые, а плавательные - они так и называются: *swimlanes*. Более формально, дорожка - часть области диаграммы деятельности, на которой отображаются только те деятельности, за которые отвечает конкретный *объект*.

Предназначены они для разбиения диаграммы в соответствии с распределением ответственности за действия. Имя дорожки может означать роль или *объект*, которому она соответствует. При использовании дорожек *нотация* слегка изменяется. Вот как, к примеру, выглядит *диаграмма* из предыдущего примера, перерисованная с использованием дорожек (рисунок 4.5).

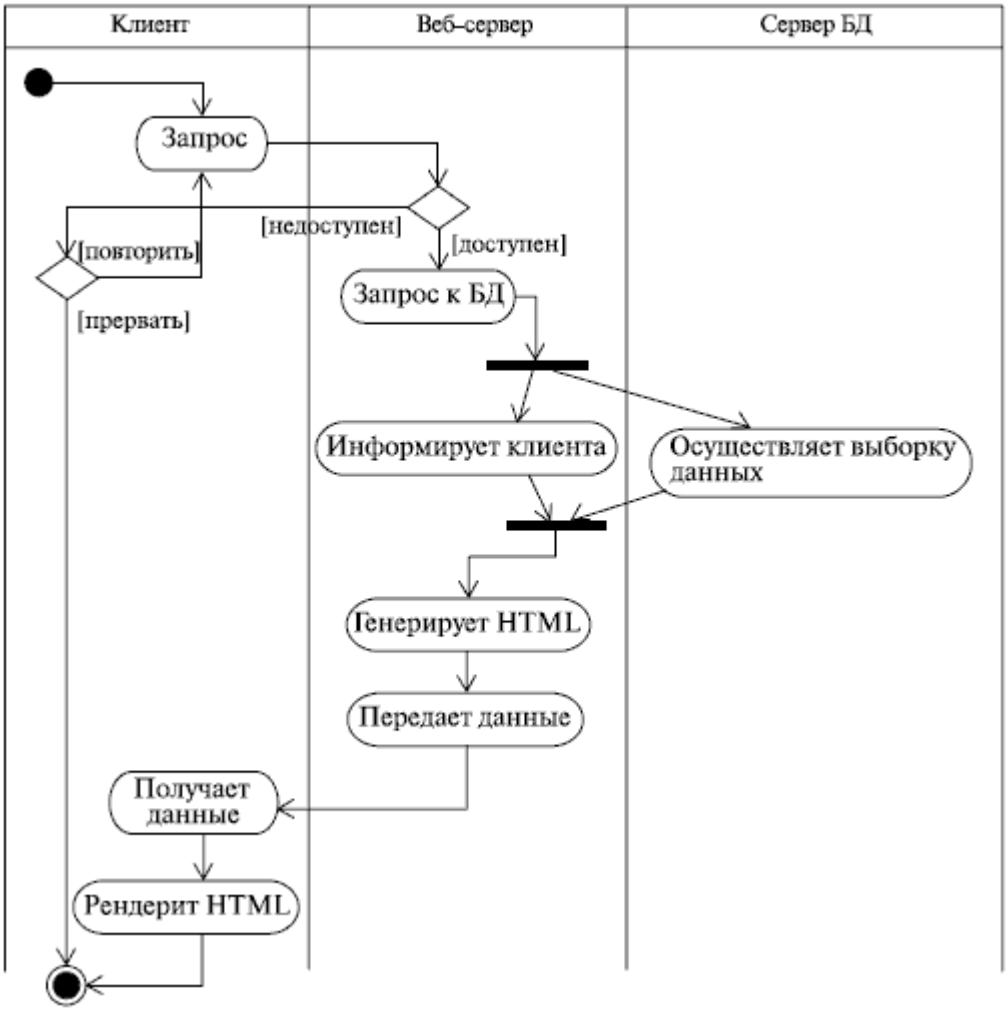


Рисунок 4.5.

Кстати, дорожки могут быть не только вертикальными, но и, если вам как автору так удобнее, горизонтальными. Изображаются горизонтальные дорожки аналогично - просто поверните "обычные" дорожки на 90 градусов против часовой стрелки!

Есть еще один нюанс нотации *диаграмм активностей*, о котором мы пока не говорили: это так называемая *траектория объекта*, или *поток объекта* (`object flow`). Суть его состоит в том, что на диаграмме деятельности можно изобразить и объекты, относящиеся к деятельности. С помощью символа зависимости (пунктирная стрелка, помните?) эти объекты можно соотнести с той деятельностью или переходом, где они создаются, изменяются или уничтожаются. Представим такую ситуацию из повседневной жизни: вы приходите в какой-нибудь фастфуд и заказываете гамбургер с колой. Что, знакомо? Во время приготовления завтрака повар создает новый *объект* - гамбургер. Пока вы нетерпеливо выпиваете колу, официант перемещает этот *объект* (подает ваш заказ). Естественно, во время завтрака вы уничтожаете этот *объект*. Вот как это выглядит на диаграмме (рисунок 4.6).

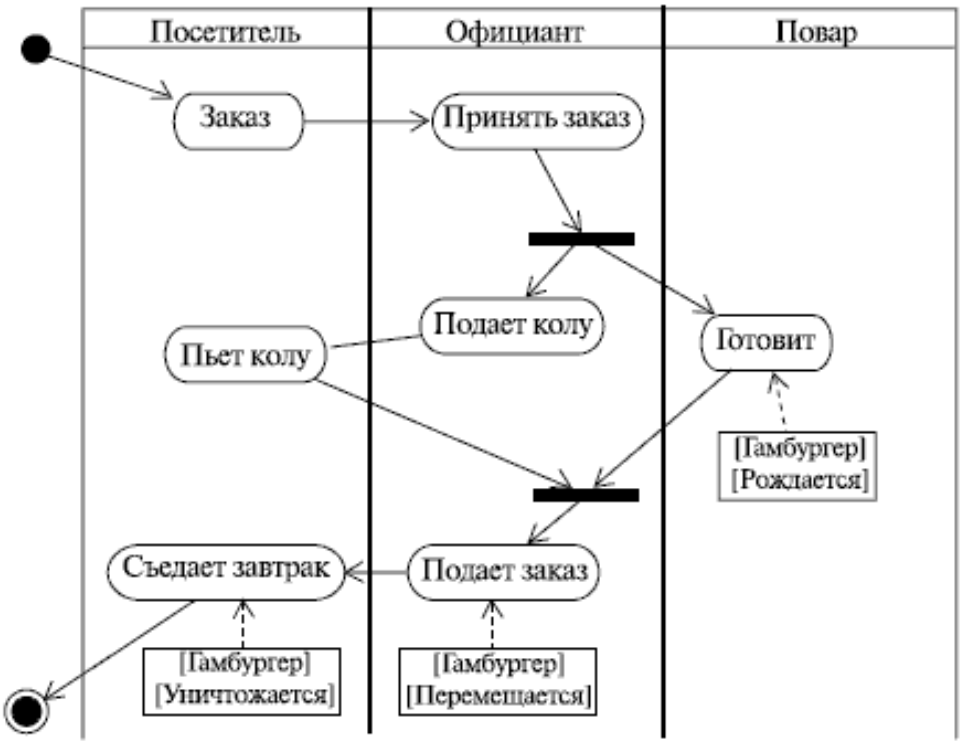


Рисунок 4.6.

На этом можно было бы и закончить наш разговор о нотации *диаграмм активностей* и их отличиях от блок-схем. Если бы не одно НО. Мы говорили, что *деятельность* - это протяженное *по* времени составное действие. Составное! То есть *составленное* из более простых действий. Вот эти-то самые простые (атомарные) действия, а вернее, последовательность их выполнения, частенько изображают внутри деятельности в виде маленькой *диаграммы активностей*. Это слегка напоминает матрешку - одна (а часто и не одна) *диаграмма* внутри другой. Мы не будем долго говорить об этом: нашей целью было просто обратить внимание читателя на подобную возможность "вложенных" диаграмм. Мы просто покажем пример, позаимствованный нами из Zicom Mentor (рисунок 4.7).

Диаграмма описывает высадку пассажиров самолета, достигших пункта назначения, и посадку новых пассажиров. Предлагаем читателю самому внимательно рассмотреть эту диаграмму. Из нее, например, можно почерпнуть, что конечных состояний может быть больше одного. Кстати, кроме начального и конечного состояний есть еще конечное состояние потока (*Flow* final *mode*).

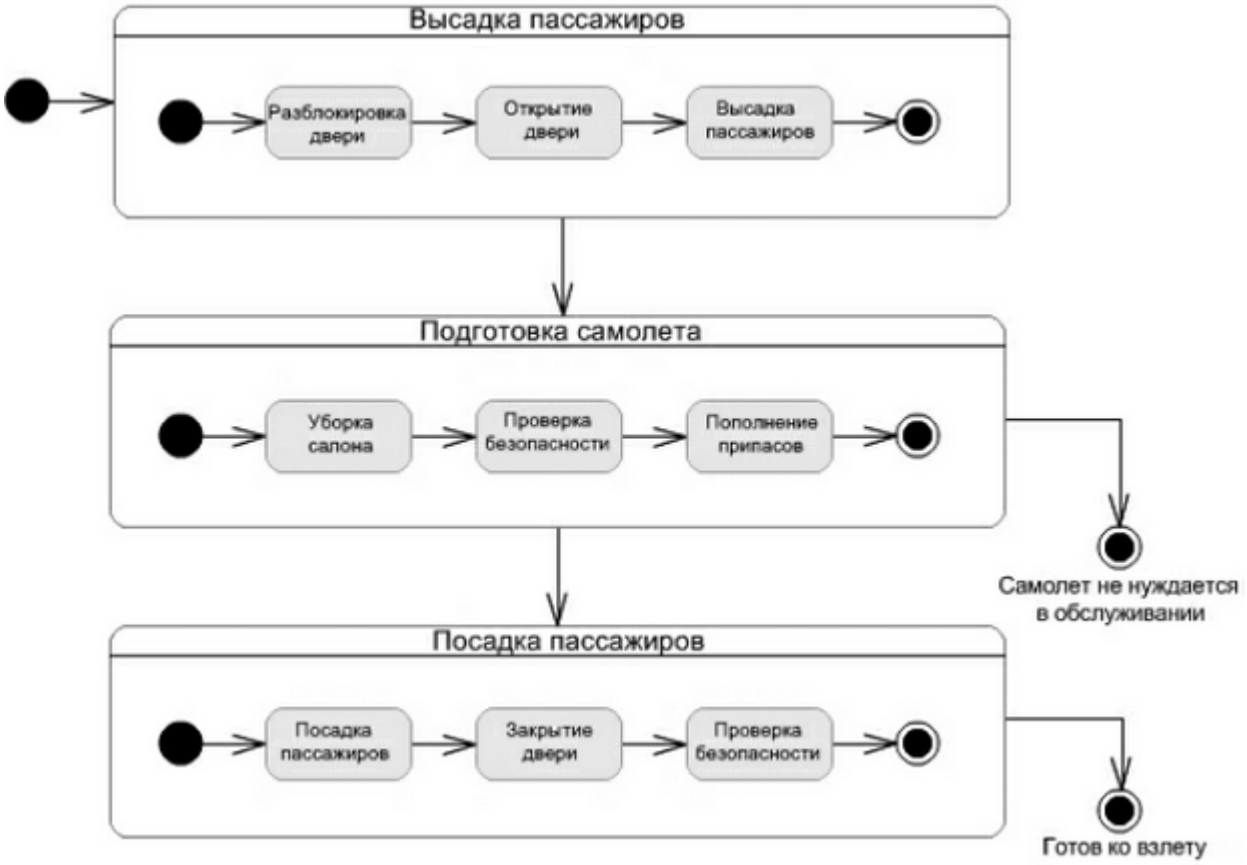


Рисунок 4.7.

От конечного состояния оно отличается вот чем: конечное состояние потока означает завершение одного потока управления, а конечное состояние говорит о завершении всех потоков управления внутри деятельности. Обозначается конечное состояние потока простым символом, напоминающим лампочку накаливания в схемах электрических цепей (рисунок4.8):

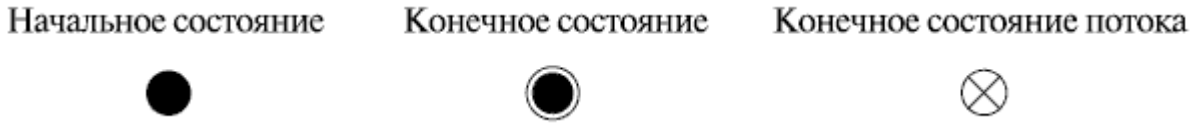


Рисунок 4.8.

Право найти примеры использования конечного состояния потока (уверяем вас, оно используется не так уж и часто), мы предоставляем читателю.

Примеры использования таких диаграмм

На практике диаграммы деятельности применяются в основном двумя способами:

1. Для моделирования процессов

В этом случае внимание фокусируется на деятельности с точки зрения экторов, которые работают с системой. Внимательный читатель, конечно же, вспомнит, что чуть ранее мы уже говорили о применимости диаграмм деятельности для описания бизнес-процессов. В случае такого использования диаграмм деятельности активно используются *траектории*

объектов. Действительно, вспомним наш пример с гамбургером: изменив роли и деятельности, легко представить на его месте некий документ. Ведь правда?

2. Для моделирования операций

В этом случае диаграммы деятельности играют роль "продвинутых" блок-схем и применяются для подробного моделирования вычислений. На первое место при таком использовании выходят конструкции принятия решения, а также разделения и слияния потоков управления (*синхронизации*).

Рассмотрим подробнее первый случай. Все мы, конечно, понимаем *бизнес-процесс* как последовательность неких действий, ведущую к достижению определенных бизнес-целей. Когда мы произносим это *слово*, в голове рождается множество ассоциаций, как то: люди, занимающие конкретные должности в управленческом аппарате (экторы), документы, которые они создают (артефакты, объекты), процесс *принятия решений* и передачи приказов *по* организационной цепочке (*управляющие* сигналы). Причем обычно все эти сущности связаны друг с другом просто невообразимым количеством явных и неявных связей, так что охватить взглядом целостную картину всего происходящего на предприятии обычно не так просто. А как же тогда все это моделируют?

Моделируют *бизнес-процессы* в несколько этапов, первым из которых является *разбиение* их на подпроцессы. Подпроцессы, являющиеся "участками большого процесса", описать легче. А там, глядишь, и составится целое из частей. Дальше выделяют ключевые объекты (и создают для них дорожки), определяют предусловия и постусловия каждого процесса (т. е. его границы), описывают деятельности и переходы, отображают на диаграммах состояния ключевых объектов, в которые они переходят в ходе процесса. Все это звучит довольно сложно, а на практике происходит еще сложнее: ведь создается не какая-то абстрактная *диаграмма*, а модель реального бизнес-процесса в реальной компании, занимающейся реальным бизнесом, где цена ошибки может быть очень высока. Чтобы окончательно не запугать читателя, приведем просто пример использования *диаграммы активностей* для описания процесса разработки *ПО* в OpenUP (рисунок 4.9).

Выглядит, конечно, не совсем так, как мы привыкли, но все же, сомнений не остается - да, это именно *диаграмма* активностей. *Нотация* слегка отличается, но все понятно и без дополнительных пояснений.

А теперь перейдем к рассмотрению моделирования операций с помощью *диаграмм активностей*. Как мы уже говорили, в этом случае *диаграмма* активностей превращается в "продвинутую" блок-схему, предоставляющую дополнительные возможности, например, *отображение* параллельно выполняющихся операций.

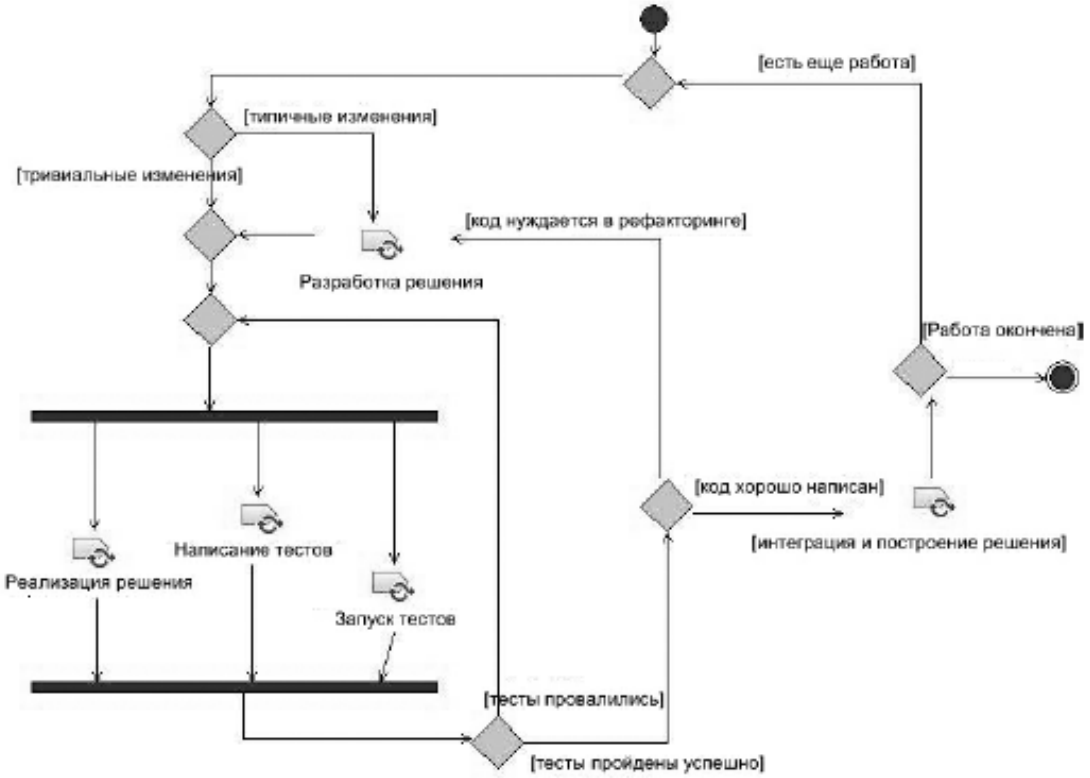


Рисунок4.9.

Возникает соблазн попытаться выполнить *кодогенерацию* такой диаграммы или даже откомпилировать ее и сразу получить выполняемый *файл*. Поспешим отметить, что вы не одиноки в таком желании - попыток создать пакет для генерации приложений непосредственно из диаграмм *UML* было предпринято множество. Некоторые даже оказались более-менее удачными - вспомним, например, Rational Rose *Real Time*. Таким образом, при моделировании операций *UML* становится языком визуального программирования!

Приведем пример моделирования одной из базовых алгоритмических конструкций, например, *цикла* с постусловием (рисунок 4.10):

Рисунок 4.10.

Советы по построению диаграмм активностей

Процесс построения *диаграммы активностей* можно описать в виде последовательности таких действий:

1. Составление перечня деятельностей в системе

Как исходные данные для этой операции хорошо подходит список прецедентов (или список операций - см. два способа использования диаграмм деятельности). Дополняться диаграммой активности может каждый сценарий использования. Можно также попытаться описать связь между ними.

2. Принятие решения о необходимости построения диаграммы деятельностей

Несмотря на то что вы уже начали работу в этом направлении, вы все же можете решить отказаться от продолжения построения диаграммы деятельностей. Причины тому могут быть различными, например, система одномоментно меняет свои состояния (как светофор) или ее поведение достаточно очевидно. (Помните пример с *циклом с постусловием*? Наверняка многие читатели подумали: "Зачем моделировать такие простые и очевидные вещи?". Теперь вы знаете зачем - чтобы показать нецелесообразность этого.)

3. Определение зависимостей между деятельностями

Для каждой активности нужно найти активности, непосредственно предшествующие (и следующие за ней тоже), то есть активности, без выполнения которых поток управления не может перейти к данной деятельности.

4. Выделение параллельных потоков деятельностей

Выделите активности, имеющие общих предшественников. Зачем - думаем, и так понятно.

5. Определение условий переходов

Сформулируйте выражения, которые могут принимать только два значения - "истинно" или "ложно", соответствующие альтернативным потокам управления. Теперь вы знаете, что писать рядом с символами принятия решений!

6. Уточните сложные деятельности

Повторите пункты 1-6 для каждой из деятельностей (при необходимости). Помните пример с посадкой/высадкой пассажиров самолета? Присмотритесь внимательно, возможно, в проектируемой вами диаграмме тоже будет нелишним применить "принцип матрешки". А как это работает на практике? Да легко! Рассмотрим, например, *моделирование* пословицы "После драки кулаками не машут":

1. Выделяем деятельности: драться, махать кулаками.
2. Следует ли строить диаграмму в этом случае? Вообще-то нет. Но ведь это пример!

3. Определяем зависимости между деятельностями: размахивание кулаками не происходит после драки.

4. Определяем параллельные деятельности: вроде бы тут таких не наблюдается...

5. Определяем условия переходов: драка состоялась? Если "нет", то машем кулаками, если "да", то нет.

6. Уточняем сложные деятельности: при драке машут не только кулаками, но и ногами. А еще можно пинаться головой и использовать подручные средства, мебель, например. Плюс можно выделить еще подготовительные деятельности (выбор места для нападения) и завершающие (вынос раненых).

А теперь попробуйте все это смоделировать. Правда, легко? Ведь все уже разложено *по* полочкам - только рисуй! А что относительно процесса построения *диаграмм активностей* говорят классики? Тот же Буч, например, писал:

Создавая диаграммы деятельности, не забывайте, что они лишь моделируют срез некоторых динамических аспектов поведения системы. С помощью единственной диаграммы деятельности никогда не удастся охватить все динамические аспекты системы. Вместо этого следует использовать разные диаграммы деятельности для моделирования динамики рабочих процессов или отдельных операций.

Выводы

1. Диаграммой деятельности можно дополнить любой элемент модели, имеющий динамическое поведение.

2. Диаграммы деятельности являются частным случаем диаграммы состояний.

3. В отличие от блок-схем, диаграммы деятельности могут отображать одновременно выполняемые действия.

4. На диаграммах активности можно использовать плавательные дорожки, распределяющие деятельности в соответствии с ролями (объектами), их выполняющими.

5. Траектория объекта позволяет показать объекты, относящиеся к деятельности, и моменты переходов этих объектов из одного состояния в другое.

6. Сложные деятельности можно дополнительно детализировать, разбив на действия и изобразив "диаграмму в диаграмме".

7. Диаграммы деятельностей можно использовать для проектирования процессов (например, бизнес-процессов) или операций (вычислений). Во втором случае UML выступает в роли визуального языка программирования.

Контрольные вопросы

1. Какие еще виды диаграмм (кроме *диаграмм активностей*) можно использовать для моделирования динамики системы?

2. Чем диаграммы деятельности отличаются от блок-схем? Какие преимущества это сулит разработчикам?
3. Что такое траектория объекта?
4. Чем конечное состояние потока отличается от конечного состояния деятельности?
5. Чем моделирование процессов отличается от моделирования операций?
6. Применимы ли диаграммы деятельности безотносительно к ООП?

Занятие 5. Диаграммы взаимодействия: крупным планом

Рекомендации *по* построению диаграмм взаимодействия.

Смысл диаграмм взаимодействия интуитивно нам, конечно же, понятен. Однако посмотрим, что о таких диаграммах говорили классики, например, Буч. **Диаграмма взаимодействия** - это диаграмма, на которой представлено взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Этот термин применяется к видам диаграмм с акцентом на взаимодействии объектов (диаграммах кооперации, последовательности и деятельности).

Несмотря на то величайшее уважение, которое мы питаем к Г. Бучу, это *определение* не кажется нам уж очень удачным. Хотя суть понятия оно передает. Наиболее важное *слово* в этом определении - это *слово* "сообщения". Действительно, как люди программирующие, мы понимаем, что взаимодействие-то как раз и состоит в обмене сообщениями между объектами! И к вопросу о сообщениях мы еще не раз вернемся. А пока же посмотрим, что Буч говорит дальше.

А дальше он объясняет, что такое диаграммы кооперации и последовательностей.

Диаграмма последовательностей - диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени.

Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

То есть *диаграмма последовательности* описывает (и именно поэтому так и называется) *последовательность*, в которой объекты отправляют и получают сообщения, а *диаграмма*кооперации - это аналог диаграммы последовательностей, который тоже показывает *обмен сообщениями* между объектами, но акцентирует внимание на *ролях*, которые объекты играют во взаимодействии. Эти два типа диаграмм вообще-то взаимозаменяемы, и решение, какую именно из них использовать в каждом конкретном случае, каждый проектировщик принимает исходя из личных предпочтений. Например, *автор* этих строк считает диаграммы последовательностей более понятным и более выразительным способом моделирования взаимодействий. Ваше мнение может быть противоположным.

А какое же *место диаграммы взаимодействия* занимают среди других диаграмм *UML*? На этот вопрос можно ответить двояко. Можно просто говорить о построении диаграмм взаимодействия как об определенном этапе в процессе моделирования. А можно вспомнить о фазах жизненного *цикла* разработки *ПО* и посмотреть, где же *диаграммы взаимодействия* окажутся в таком случае. Да, кстати, кто помнит, какая *диаграмма UML* наилучшим образом подходит для описания процессов? Хм, что-то не видно леса рук... Ах да, видим одну руку - девушка, сидящая в дальнем углу зала, за колонной... Правильно! *Диаграмма*

активностей. Что ж, попробуем нарисовать *диаграмму активностей*, описывающую процесс построения модели системы. Вот вариант такой диаграммы, предложенный одним из наших студентов (рисунок 5.1):

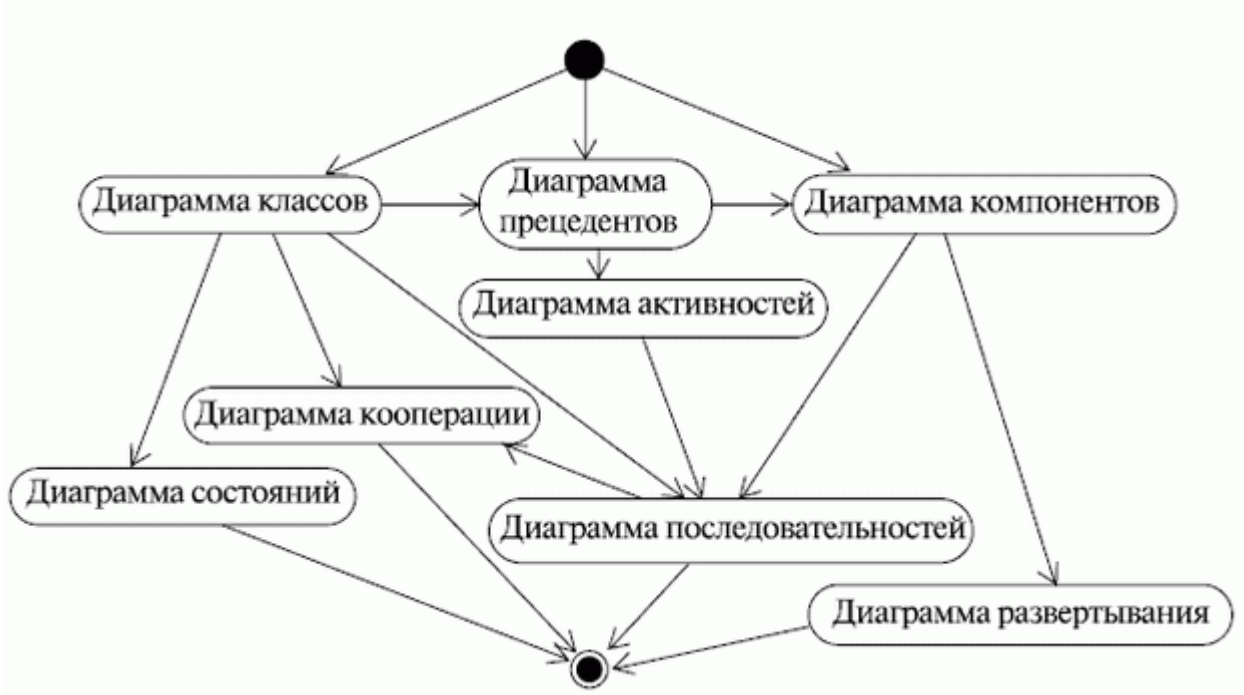


Рисунок 5.1.

М-да, не совсем *диаграмма* и не совсем активностей. Но все же она показывает то, что мы хотели показать, а именно, что *диаграммы взаимодействия* строятся после того, как описана *структура системы* (*диаграмма* классов, *диаграмма* компонентов), способы ее взаимодействия с внешним миром (*диаграмма прецедентов*) и алгоритмы действий, выполняющихся в системе (диаграмме активностей). Это как бы последний штрих, уточнение того, как именно ведет себя система путем изображения взаимодействия объектов внутри ее.

Для того же, чтобы показать *место* диаграмм взаимодействия в жизненном цикле разработки *ПО*, нарисуем еще одну "псевдодиаграмму". Правильнее было бы сказать, что та *диаграмма*, которую вы сейчас увидите (рисунок 5.2), показывает, какие артефакты разработки документируются какими диаграммами.

Рисунок 5.2.

И опять все вроде бы логично - мы строим *диаграммы взаимодействия* во время анализа поведения системы. Кстати, из рисунка (сказать "*диаграмма*" язык не поворачивается) очень хорошо видно, что *диаграмма* последовательностей и *диаграмма* кооперации взаимозаменяемы и являются альтернативными друг другу шагами процесса.

Диаграммы последовательностей и их нотация

Вступительная часть наконец-то закончилась, и мы с полным правом можем перейти к рассмотрению нотации диаграмм взаимодействия. Начнем с диаграмм последовательностей. Итак, мы уже говорили, что *диаграмма* последовательностей показывает последовательность, в которой объекты в процессе взаимодействия обмениваются сообщениями. Но как же сами объекты изображаются на такой диаграмме? А изображаются они точно таким же способом, каким мы пользовались ранее. Т. е. *объект* - это просто *прямоугольник*, внутри которого указаны подчеркнутые *имя объекта* и название класса (не обязательно), разделенные двоеточием. Объекты располагаются в верхней части диаграммы друг за другом. А вниз от каждого объекта тянется пунктирная линия, которую называют *линией жизни объекта*. Линия жизни объекта - это линия, которая изображает существование объекта на протяжении некоторого промежутка времени, и чем длиннее линия, тем дольше существует *объект*. Сообщения, которыми обмениваются объекты, изображаются в виде стрелок, направленных от линии жизни одного объекта к линии жизни другого. Линии жизни объектов, тянущиеся вниз, играют роль шкалы времени, так что сообщения, отправленные ранее, расположены выше, чем отправленные позже. Таким образом, последовательность сообщений легко читается "сверху вниз". Чуть позже мы еще вернемся к обсуждению сообщений и поговорим о том, каких видов они бывают и как их различить на диаграмме последовательностей. А пока же убедимся, что мы одинаково понимаем сам термин "сообщение": мы рассматриваем сообщение как спецификацию передачи информации от одного объекта к другому. *Объект* отправляет сообщение в расчете на то, что оно вызовет некую реакцию и за этим последует некоторая *деятельность*.

Еще одна вещь, которую можно увидеть на диаграммах последовательностей - это длинные прерывистые полосы на линиях жизни. Таким образом обозначаются периоды времени, когда *объект* имеет *фокус управления*, т. е. выполняет некоторое действие (причем неважно как - непосредственно или путем вызова некоей подчиненной *операции*).

Фокус управления на диаграммах последовательностей часто не изображают: ведь и так понятно, где он должен располагаться, достаточно взглянуть на положение стрелок, изображающих сообщения. Рисовать фокус или нет - дело привычки каждого проектировщика. Впрочем, многие средства *UML*- моделирования рисуют фокус автоматически, так что чело-

веку не нужно заботиться о его изображении. Если *объект* в процессе взаимодействия разрушается, этот факт помечают на его линии жизни крестиком, который, собственно, эту линию и заканчивает. Да, все мы смертны. Иногда так и тянется рука написать "R.I.P." рядом с таким крестиком...

Не полагаясь на выразительную силу и образность наших описаний, все же покажем примеры всех этих обозначений (рисунок 5.3).

Рисунок5.3.

А вот еще парочка обозначений. Первое из них - это *анонимный эктор*, которого изображают, если нужно показать использование объектов системы некоей *внешней сущностью* или абстрактным пользователем. Второе - это *рефлексивное сообщение*. Помните, что такое рефлексия? Правильно, самосозерцание! Тут, в принципе, происходит нечто подобное: *объект*посылает сообщение самому себе. Так рисуют, если нужно показать действие, выполняемое самим объектом (или внутри него), либо то, что *объект* сам себя вводит в некоторое состояние (рисунок 5.4).

Рисунок 5.4.

И еще одно - мы легко можем представить ситуацию посылки сообщения в зависимости от истинности некоторого условия. Например, если цена приглянувшейся нам в магазине вещи меньше ста условных единиц, мы вполне можем приобрести ее за наличные. Покупку на сумму от 100 до 1000 долларов можно оплатить кредитной картой, а чтобы купить нечто, стоящее дороже 1000 у. е., придется брать *кредит*. А как изобразить такие ситуации (*ветвления*) на диаграмме последовательностей? Да легко (рисунок 5.5)!

Рисунок 5.5.

Впрочем, *ветвление* - конструкция для диаграмм последовательностей непопулярная и используется она в них очень редко. Считается, что ветвления более присущи диаграммам деятельностей...

Ранее мы говорили, что сообщение посылается объектом в расчете на определенную реакцию, на то, что за этим последует некоторая *деятельность*. Например, посылка *ответного сообщения*. А как на диаграммах последовательностей изображаются ответные сообщения? Обычно их изображают пунктирной линией со стрелкой, хотя часто они имеют точно такой же вид, как и обычные сообщения, только направлены в противоположную сторону. Как именно их рисовать - пунктирной линией или сплошной - решать вам. Это абсолютно не принципиально (рисунок 5.6).

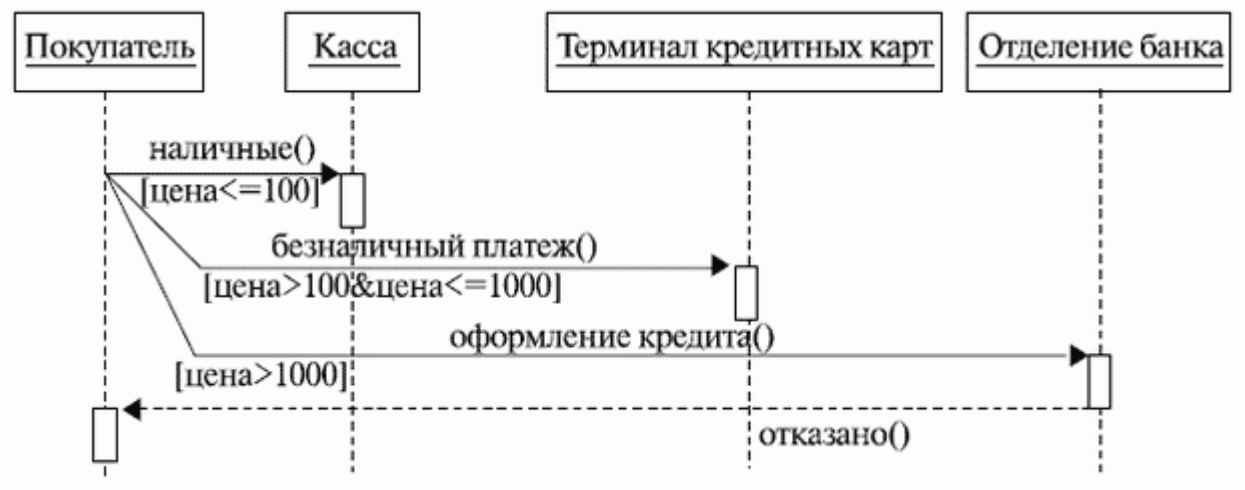


Рисунок 5.6.

Хм, картина усложняется. Мы уже видели два вида стрелок. И соответственно, два вида сообщений - прямое и ответное. Может быть, есть еще какие-то виды сообщений, о которых мы пока не знаем? Да, есть. Сами *по* себе сообщения бывают синхронными и асинхронными. *Синхронные* сообщения приостанавливают *поток* выполнения до тех пор, пока не будет получен ответ. Все сообщения, которые мы рассматривали в наших примерах, были именно синхронными. Пусть мы и не везде рисовали ответное сообщение, но оно подразумевалось: банк выносит решение о предоставлении кредита и сообщает его вам, *терминал* кредитных карт подтверждает транзакцию и печатает чек, на котором вы ставите подпись, кассир выдает вам подтверждение платежа - кассовый чек. Синхронные сообщения изображаются сплошной линией с треугольной закрашенной стрелкой на конце.

Другой вид сообщений - *асинхронные* сообщения. Они не ждут ответа, не приостанавливают *поток* выполнения - сразу после их посылки происходит немедленный переход к следующему шагу, и последовательность продолжается. Входя в *офис* поутру и говоря коллегам "hello, how are you?", вы ведь не ждете, что они остановят вас и начнут в течение часа рассказывать о своих проблемах? Это просто формальное приветствие, не предусматривающее ответа (асинхронное). Асинхронные сообщения изображаются сплошной линией с обычной (составленной из двух отрезков) стрелкой на конце.

А как изображаются ответные сообщения, мы уже знаем (рисунок 5.7).

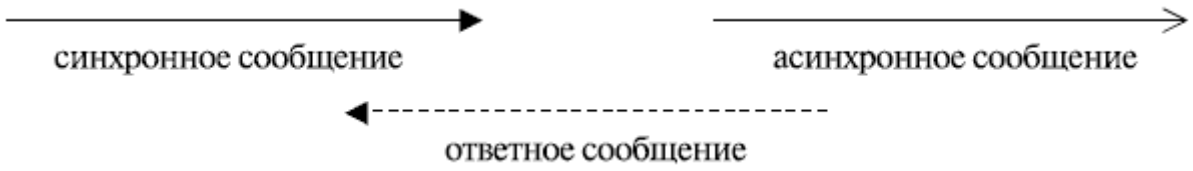


Рисунок 5.7.

Возможны случаи, когда нам известен адресат сообщения, но неизвестен его отправитель. С примерами таких сообщений (в бумажном виде) в советские времена довольно часто встречались секретари госучреждений. Такие сообщения называют *найденными*. Или *обратный* случай: отправитель известен, а получатель - нет. Пример? Да хотя бы записки, *запечатанные* в бутылки, которые когда-то бросали в море жертвы кораблекрушений! Такие сообщения называют... Да-да, именно - *потерянными*.

На диаграммах они изображаются без особых изысков (рисунок 5.8).

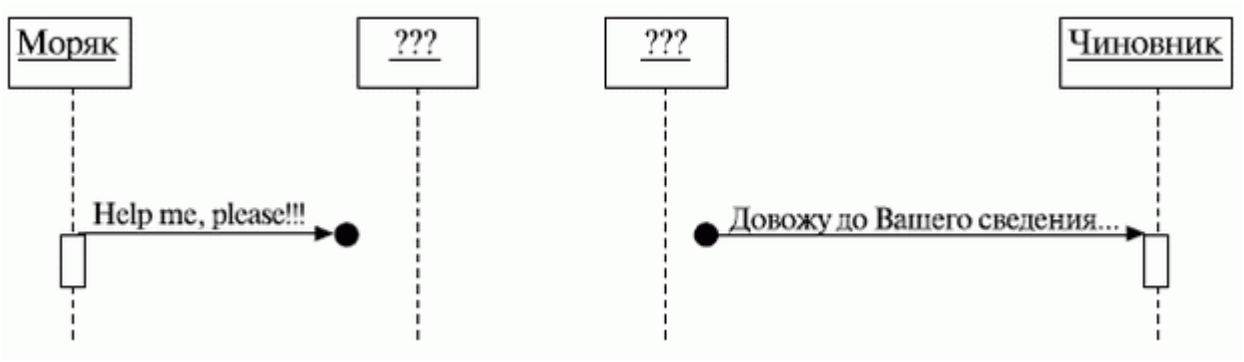


Рисунок 5.8.

Рассмотрим, наконец, "полный" пример диаграммы последовательностей. И конечно же, этот пример мы возьмем с сайта шуток на *UML* http://www.umljokes.com (рисунок 5.9).

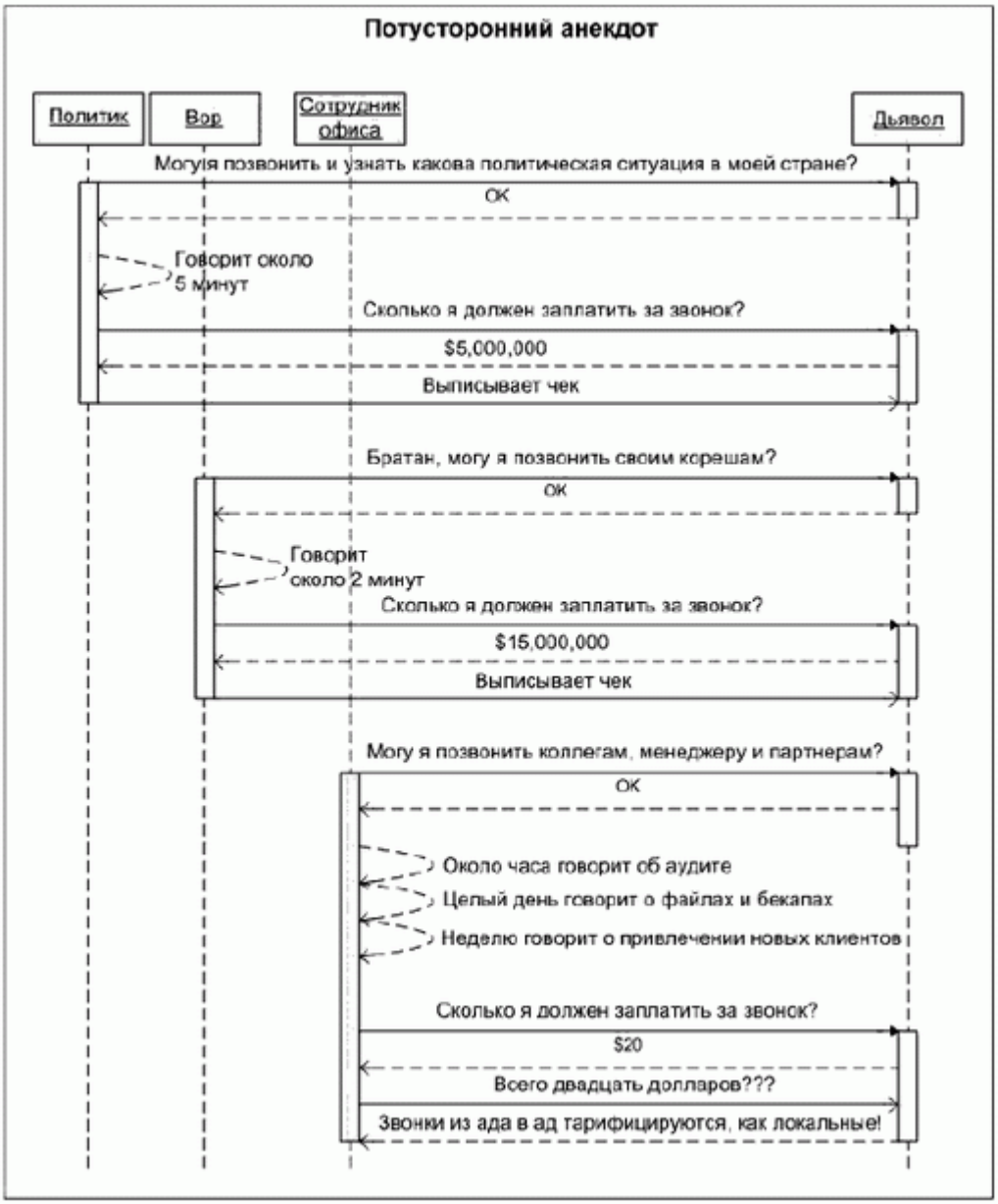


Рисунок 5.9.

Не правда ли, очень жизненный анекдот? А вот еще один пример, показывающий, что, задав вопрос "сколько будет два плюс два?", вы не всегда услышите в ответ "четыре". Ответ

на любой вопрос всегда сильно зависит от личности, настроения, уровня интеллекта отвечающего, даже от его профессии. И вот вам тому *доказательство* (рисунок 5.10).

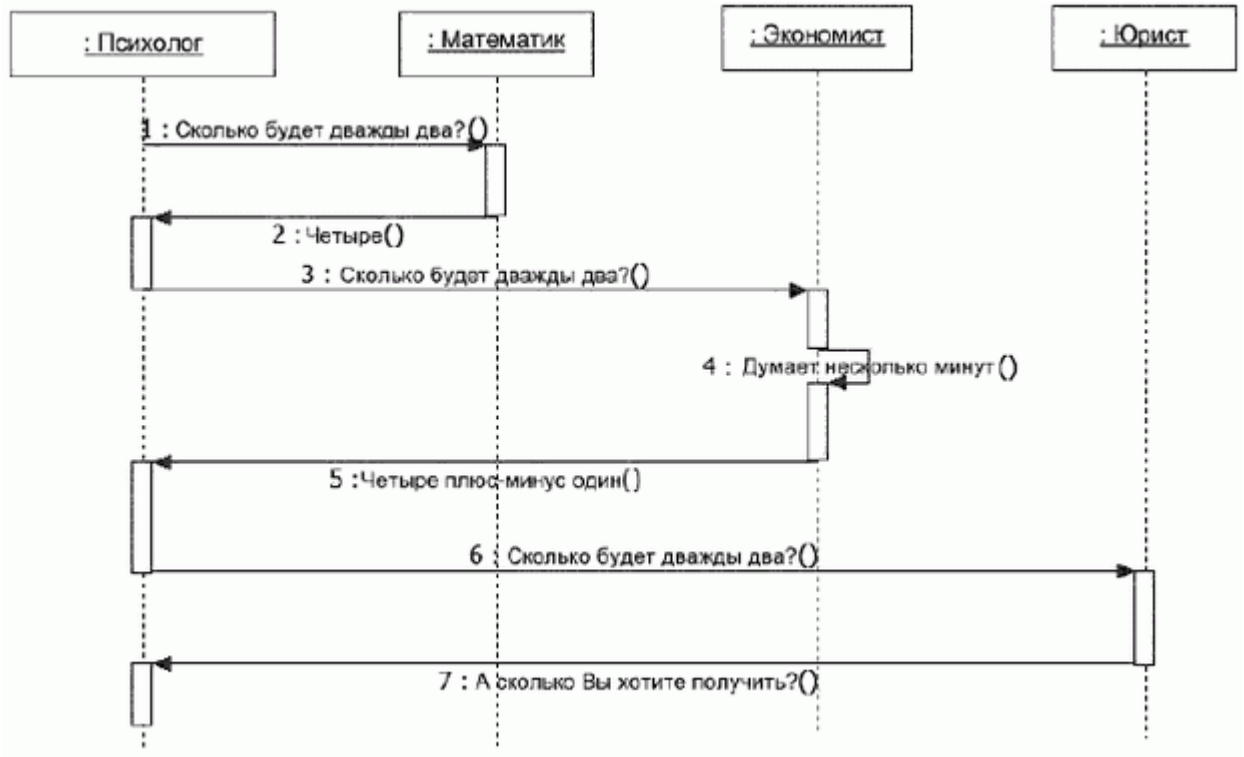


Рисунок 5.10.

Диаграммы кооперации и их нотация

Что ж, один из видов диаграмм взаимодействия, а именно диаграммы последовательностей, мы рассмотрели. Перейдем же к рассмотрению альтернативы - диаграммам кооперации. Собственно, *слово* "*кооперация*" значит "совместная *деятельность*", "сотрудничество". Такие диаграммы показывают, как объекты работают вместе для достижения общей цели, акцентируя на их ролях.

Следует отметить, что здесь имеет *место* некоторая терминологическая путаница. В оригинале такие диаграммы называются *Collaboration Diagram*. *Слово* "*collaboration*", конечно же, *синоним* слова "cooperation", но в "русском" варианте звучит хуже. Поэтому в русскоязычных учебниках говорят "*диаграмма* кооперации", а не "коллаборации". Кроме этого, само это название немного устарело - в *UML* 2.x подобные диаграммы называются **Communication Diagram**. Впрочем, все три слова - "cooperation", "*collaboration*" и "*communication*" - являются синонимами, так что довольно часто используется "старое" название. Часто даже, говоря *"диаграммы взаимодействия", подразумевают именно диаграммы кооперации*.

Итак, мы уже говорили, что, подобно диаграммам последовательностей, диаграммы кооперации предназначены для описания динамических аспектов моделируемой системы. Обычно они применяются для того, чтобы:

- показать набор взаимодействующих объектов в реальном окружении "с высоты птичьего полета";
- распределить функциональность между классами, основываясь на результатах изучения динамических аспектов системы;
- описать логику выполнения сложных операций, особенно в тех случаях, когда один объект взаимодействует еще с несколькими объектами;
- изучить роли, выполняемые объектами внутри системы, а также отношения между объектами, в которые они вовлекаются, выполняя эти роли.

Говоря о диаграммах кооперации, часто упоминают два "уровня" таких диаграмм - *уровень экземпляров* (примеров, Instance-Level) и *уровень спецификации* (Specification-Level). В чем же разница? Ответ прост: уровень экземпляров отображает взаимодействия между объектами (экземплярами классов); такая *диаграмма* обычно создается, чтобы исследовать *внутреннее устройство* объектно-ориентированной системы. Уровень же спецификации используется для изучения ролей, исполняемых в системе основными классами. В любом случае, *диаграмма* взаимодействия не отображает процесс. Она показывает взаимодействие между объектами, которое, как мы уже знаем, осуществляется путем посылки и приема сообщений. При этом точная последовательность сообщений не так хорошо видна, как на диаграмме последовательностей, так что если для вас важно отобразить именно порядок отправки и приемов сообщений, используйте диаграмму последовательностей.

Так, все вступительные слова сказаны, теперь поговорим о нотации диаграмм взаимодействия. На диаграммах взаимодействия вы можете увидеть... нет, не угадали, не почтальонов, которые сломя голову мечутся между объектами, спеша доставить сообщения в бумажных конвертах с большими сургучными печатями. На **диаграммах взаимодействия вы увидите объекты, классы, сообщения, связи и кооперации. Но обо всем *по* порядку.**

Итак, *кооперация* (*collaboration*). Это статическая конструкция для моделирования набора сущностей, взаимодействующих друг с другом. *Кооперация* определяет набор взаимодействующих ролей, используемых вместе, чтобы показать некую функциональность. *Кооперация* часто реализует некоторый *паттерн (шаблон проектирования)*. Впрочем, о шаблонах проектирования мы сейчас говорить не будем, поскольку они выходят за рамки этого курса и первого теста программы OCUP. Заинтригованным читателям мы предложим попробовать ввести *словосочетание* "*design patterns*" в адресную строку браузера. Спорим, попадете на статью "*Design pattern* (*computer science*)" из "Википедии"?

Кооперация изображается в виде *эллипса с пунктирной границей*, причем символ этот может использоваться двумя способами. Вот первый способ (рисунок 5.11):

Рисунок 5.11.

Мы видим, что эта *диаграмма* буквально иллюстрирует наши слова о кооперации как наборе ролей, используемых вместе, чтобы показать некую функциональность, в данном случае - выполнение ежемесячного резервного копирования. Второй способ показывает прикрепленные к объектам (классам) роли в рамках данной кооперации. Назначение роли изображается пунктирной линией со стрелкой на конце, направленной в сторону объекта. *Имя роли* указывается на конце линии, рядом с объектом. Посмотрите, например, на эту диаграмму (рисунок 5.12):

Рисунок 5.12.

Все ведь понятно, правда? Видно, кто какую роль играет и в каком взаимодействии (кооперации). А еще показана генерализация и кооперации, и самих исполнителей.

С кооперацией разобрались. Отметим, что, скорее всего, в реальном моделировании вы с ней будете встречаться крайне редко. Следующие элементы, которые можно увидеть на диаграмме взаимодействия - это *объекты и классы*.

Поскольку *диаграмма* кооперации - всего лишь альтернативная форма представления той же информации, которая содержится в диаграмме последовательностей, то и обозначения объектов (классов) в ней, *по* сути, такие же, как и на диаграмме последовательностей (и

на других диаграммах). Чтобы проиллюстрировать это утверждение, приведем пример *диаграммы взаимодействия*, позаимствованный нами с сайта http://www.agilemodeling.com/ (а точнее, http://www.agilemodeling.com/style/collaborationDiagram.htm) (рисунок 5.13):

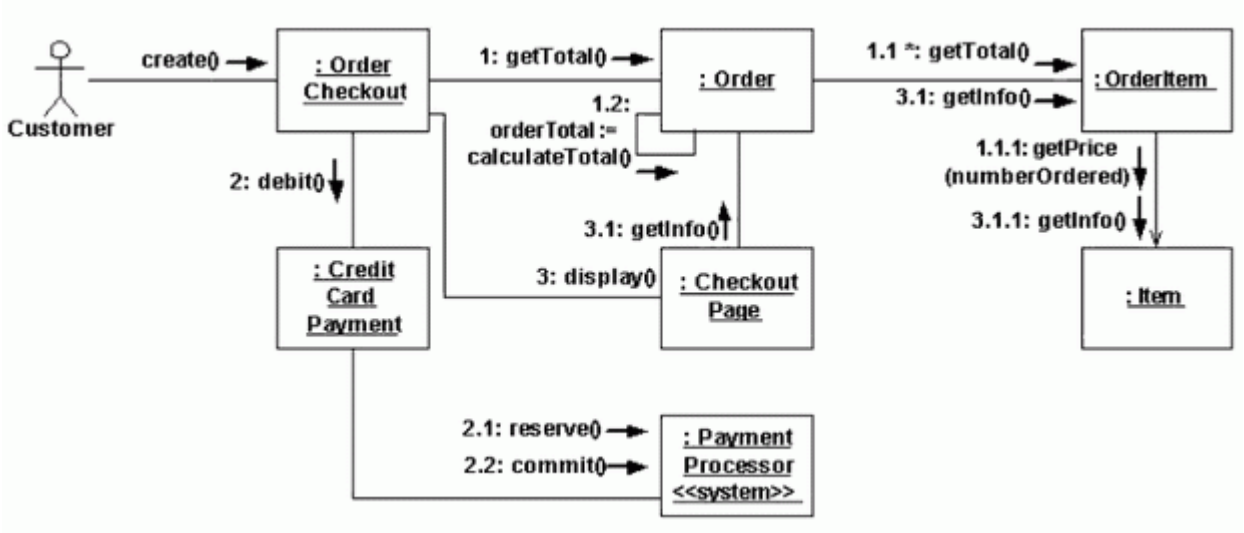


Рисунок 5.13.

Как видите, *диаграмма* иллюстрирует покупку некоторого товара (вероятно, в онлайне) и оплату с помощью кредитной карты. Еще одна интересная вещь, которую можно увидеть на этой диаграмме - это *сообщения*, вернее, то, как они изображаются. Сообщения показаны в виде текста (названия метода) со стрелкой. Но есть один нюанс: на диаграмме последовательностей было легко показать последовательность отправки сообщений, так как линии жизни служили одновременно "осями времени", направленными вниз, и, естественно, было видно, что нижние сообщения отправлены позже верхних.

В диаграммах взаимодействия проблему отображения очередности сообщений решили просто - перед названием каждого сообщения просто пишут его номер. Выглядит эта конструкция так: *номер:название_сообщения*.

Причем часто используют и *составные номера*. Например, *объект* отправил другому объекту сообщение с номером 1. Когда *объект*-получатель в свою *очередь* отправляет сообщения другим объектам, они получают номера 1.1, 1.2 и т. д.

Иногда нужно показать одновременную отправку сообщений. Чтобы отметить параллельные потоки сообщений, их номера предваряют буквами A, B, C, D и т. д.

Вот пример таких обозначений, позаимствованный опять-таки с http://www.agilemodeling.com/ (рисунок 5.14):

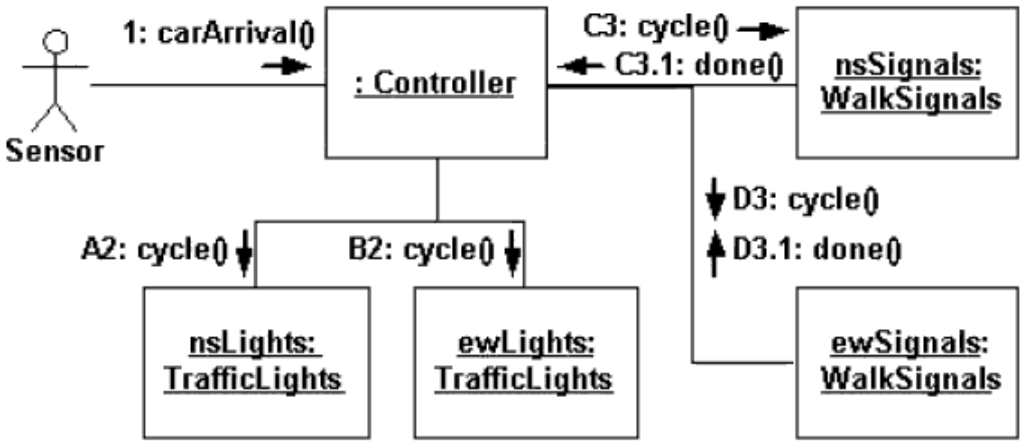


Рисунок 5.14.

Еще одна "мелочь", на которую хотелось бы обратить внимание пользователя - это *мультиобъекты*. *Мультиобъект* показывает, что на "дальнем" конце ассоциации находится не один, а *целый набор* объектов. Такая конструкция используется, чтобы показать операцию, которая нацелена на *целый* набор объектов. *Мультиобъект* изображается как два прямоугольника, смещенных *по* отношению друг к другу, что создает впечатление "колоды карт". Сообщение, отправленное мультиобъекту, означает сообщение к набору объектов, например, операция выбора - поиска определенного объекта. Пример подобной диаграммы показан на рисунке (рисунок 5.15):

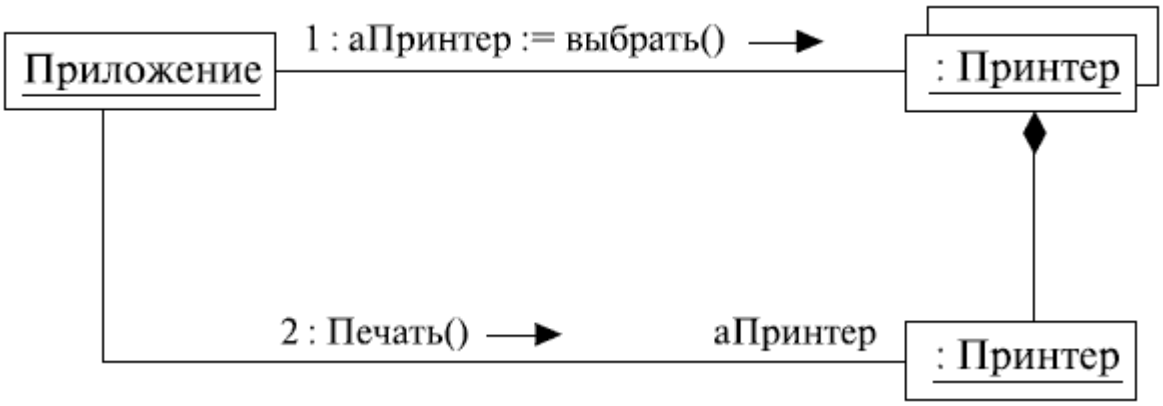


Рисунок 5.15.

Смысл диаграммы вполне понятен: для печати документа из некоторого приложения необходимо выбрать из всех доступных некий конкретный принтер. Символ композиции применен для того, чтобы показать, что принтер входит в состав набора объектов. Предположим теперь, что у нас доступны несколько сетевых принтеров и один локальный. Как показать, что выбран именно он? Легко. Для этого используют *связи со стереотипами*. Чтобы показать, что выбран *локальный принтер*, чуть изменим предыдущую диаграмму (рисунок 5.16):

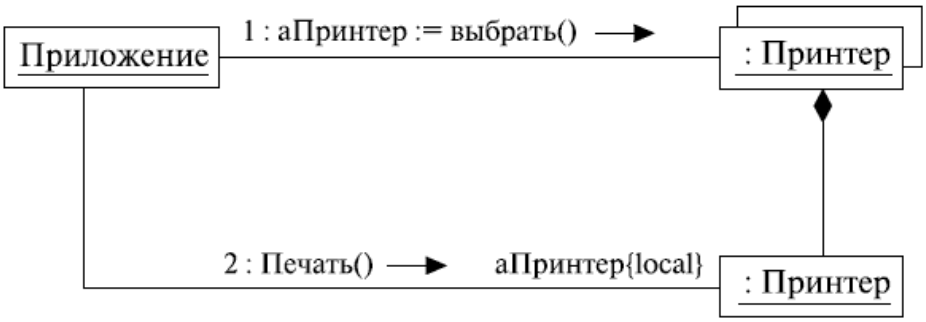


Рисунок 5.16.

Следует отметить, что иногда вместо фигурных скобок используются угловые кавычки (как мы привыкли делать, указывая стереотип в названии компонента или класса), но чаще все же применяют *фигурные скобки*. Измененная *диаграмма* стала еще более понятной, не правда ли? Чтобы закрепить полученные знания о связях со стереотипами, приведем еще один пример (рисунок 5.17):

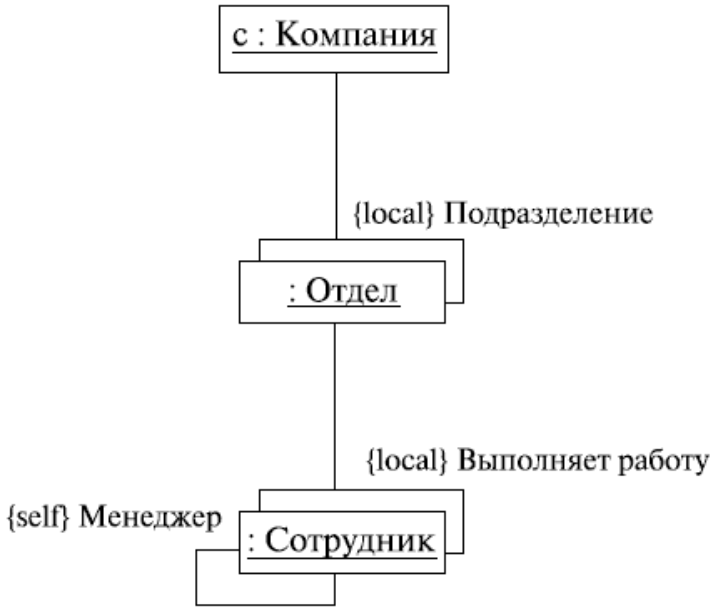


Рисунок5.17.

Смысл диаграммы опять вполне понятен, ведь правда? А стереотипы связей позволяют исключить неоднозначности, которые могли бы быть, если бы мы говорили, например, о многонациональной распределенной компании...

И еще одна вещь, которая связана с понятием кооперации - *композитный объект*. Композитный *объект* - это высокоуровневый *объект*, состоящий из нескольких частей-объектов. Это экземпляр композитного класса, реализующего композитное *агрегирование* класса и его частей. Композитный *объект* - понятие, *близкое к понятию кооперации*, но

более простое и более ограниченное. Это конструкция, показывающая целое в виде взаимодействующих частей, но в основном с точки зрения композиции. Изображается композитный *объект* в виде прямоугольного символа объекта, но с некоторыми отличиями:

- имя объекта указывается в верхней части прямоугольника, отделенной от его остальной части горизонтальной линией;
- в нижней части прямоугольника размещаются части композитного объекта, также, естественно, изображаемые символами объектов;
- части композитного объекта могут (и даже должны) быть связаны между собой;
- допускается ситуация, когда некоторые части композитного объекта сами являются композитными объектами.

Посмотрим же, как это выглядит на примере (рисунок 5.18).

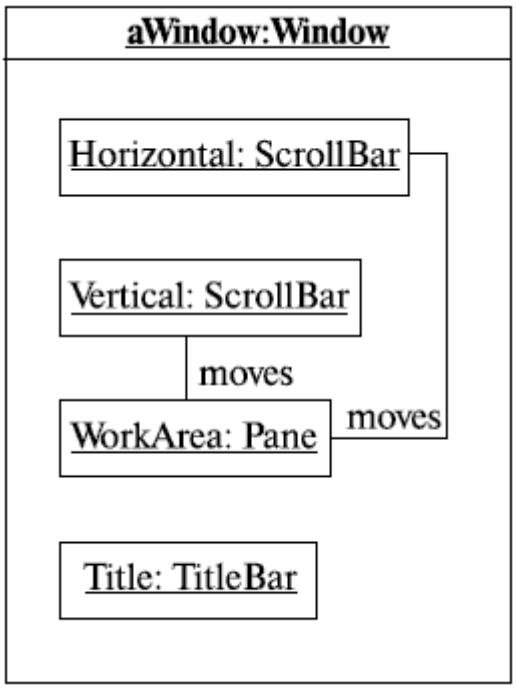


Рисунок 5.18.

Не правда ли, эта упрощенная модель графического окна проста и понятна? Окно имеет заголовок, рабочую область и две полосы прокрутки - горизонтальную и вертикальную, которые ее перемещают. Все просто!

Композитный *объект* лишь близок *по* значению к кооперации, но не встречается на диаграммах взаимодействия "в чистом виде". На диаграммах взаимодействия иногда можно увидеть очень близкую *по* смыслу конструкцию, а именно *активный объект*. Активными называют объекты, которые владеют собственным потоком управления и могут инициировать выполнение действий. *Пассивные объекты* содержат данные, но не могут инициировать выполнение. Конечно, пассивные объекты могут посылать сообщения в процессе обработки полученных запросов. *Активный объект* (или, вернее, его роль) выглядит на диаграмме как

прямоугольный символ объекта, но с утолщенными границами. Часто *активный объект* изображается как композитный *объект*, содержащий объекты-части.

Посмотрите, например, на эту диаграмму, позаимствованную нами chttp://etna.intevry.fr/COURS/UML/notation/notation8a.html (рисунок 5.19):

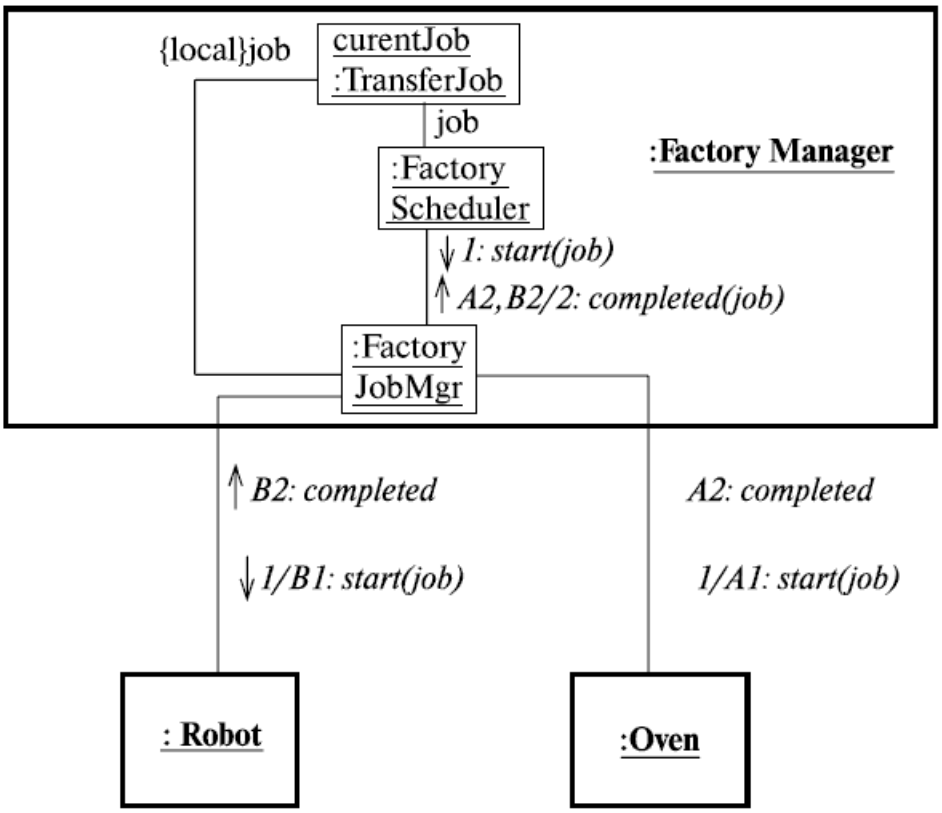


Рисунок 5.19.

Как видите, все объекты, отображенные на диаграмме, являются активными. Таких объекта три - это робот, печь и *менеджер* цеха, который изображен как композитный *активный объект*.

Вообще же, если говорить о композитных объектах, то следует отметить, что в *UML* 2 появился новый тип диаграмм - **композитная структурная диаграмма**. Она показывает внутреннюю структуру элемента, включая точки взаимодействия с другими частями системы. Таким образом, отображается состав и отношения между частями, которые совместными усилиями реализуют поведение элемента. Вот отличный пример композитной диаграммы из Zicom Mentor (рисунок 5.20).

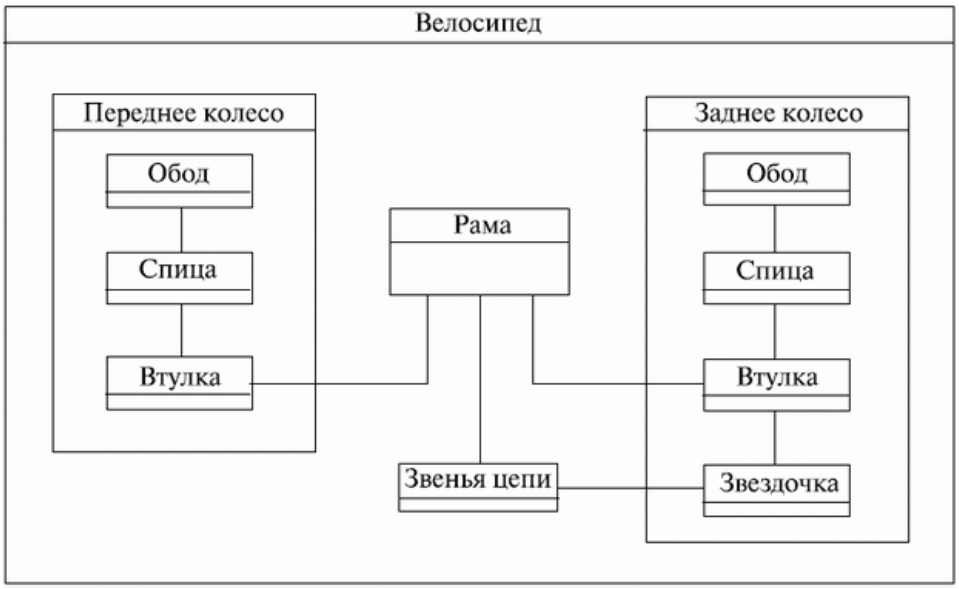


Рисунок 5.20.

Прекрасная модель велосипеда! Узнаете старых знакомых - композитные объекты?

К сожалению, композитные структурные диаграммы находятся за пределами тематики экзамена UM0-100, поэтому больше о них мы здесь говорить не будем. Однако напоследок скажем, что, кроме внутренних частей, на таких диаграммах можно увидеть еще одно новшество *UML* 2 - *порты*. *Порт* - это типизированный элемент, который представляет *"видимую снаружи"* часть содержащего его элемента. *Порт*, как это и следует из названия, определяет взаимодействие элемента модели с окружающей его средой. *Порт* может размещаться на границе части, класса или композитной структуры. *Порт* может описывать сервисы, предоставляемые элементом модели (и требуемые окружающей его средой). Изображается *порт*как именованный (недаром же мы ранее сказали "типизированный") *прямоугольник* на границе содержащего его элемента модели (впрочем, иногда можно увидеть символ порта и внутри символа класса - тогда говорят, что *класс* имеет *скрытый порт*). Чтобы покончить с этими отступлениями от темы, покажем, как все это выглядит на диаграмме, и вернемся к диаграммам взаимодействия (рисунок 5.21).

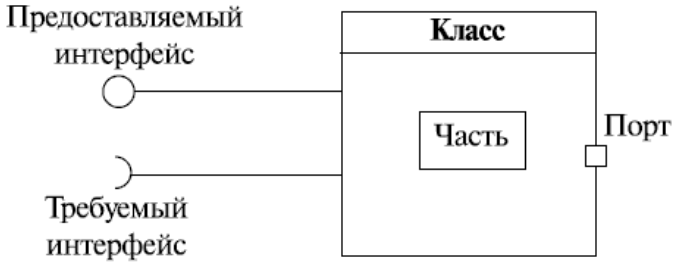


Рисунок 5.21.

Вспомнили, что изображают "леденцы на палочке"? Правильно, интерфейсы - предоставляемый классом и требуемый им, молодцы. А теперь вернемся к основной нити нашего повествования.

Рекомендации по построению

Каким же образом и в какой последовательности следует действовать, чтобы построить качественную диаграмму взаимодействия? Начинать нужно с выделения тех и только тех классов, объекты которых участвуют в моделируемом вами взаимодействии. Затем все объекты наносим на диаграмму. Следует также определить те объекты, которые будут существовать постоянно, и те, которые будут существовать только во *время выполнения* ими действий в рамках моделируемого взаимодействия.

Так, объекты нарисованы, можно переходить к сообщениям. Возможно, чтобы лучше передать роли, играемые объектами во взаимодействии, понадобится использовать различные разновидности сообщений и стереотипы. Для уничтожения объектов, которые существуют только во *время выполнения* неких действий, тоже нужно предусмотреть специальные сообщения.

А если есть ветвления? Самые простые случаи ветвлений процесса взаимодействия можно показать и на одной диаграмме - помните пример с разными способами платежа в зависимости от стоимости приглянувшейся вещи? Но при изображении ветвлений *диаграмма* становится более сложной для понимания "с лету". Нужно соблюдать баланс между детализацией и сложностью: лучше каждый альтернативный *поток* управления показывать на отдельной диаграмме. В таком случае следует рассматривать такие "частные" диаграммы в комплексе, как одну модель взаимодействия.

Если же хочется еще больше детализировать диаграмму, можно ввести временные ограничения на выполнение отдельных действий. Впрочем, для простых асинхронных сообщений временные ограничения, скорее всего, не нужны. А вот необходимость синхронизации сложных потоков управления часто требует использования таких ограничений. *Запись* их должна следовать правилам языка объектных ограничений (OCL, *Object Constraint Language*). Рассмотрение OCL выходит за рамки нашего курса и экзамена UM0-100, для подготовки к которому он написан. Хотя, сами того не зная, мы уже использовали OCL - вспомните условия в квадратных скобках под сообщениями на диаграмме последовательностей с ветвлением!

Выводы

1. Диаграмма последовательностей - диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени.
2. Диаграмма кооперации - альтернативная форма представления информации, содержащейся в диаграмме последовательностей.
3. Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

4. Существуют различные типы сообщений: синхронные, асинхронные и ответные, потерянные и найденные.

5. Диаграммы кооперации бывают двух "уровней" - уровня экземпляров и уровня спецификации.

6. Кооперация - это статическая конструкция для моделирования набора сущностей, взаимодействующих друг с другом.

7. С диаграммами кооперации связаны такие понятия, как *мультиобъекты*, композитные объекты и активные объекты.

Контрольные вопросы

1. Может ли диаграмма последовательностей содержать объект с линией жизни, но без фокуса управления?

2. Чем отличаются представления кооперации на уровне спецификации и на уровне примеров?

3. В чем разница между активными и пассивными объектами?

4. Чем асинхронное сообщение отличается от синхронного?

5. Что такое мультиобъект?

6. Что такое композитный объект и как он связан с понятием кооперации?

7. Как можно избежать усложнения диаграммы взаимодействия с разветвленным потоком управления?

Занятие 6. Диаграммы прецедентов: крупным планом

Несколько слов о требованиях

Итак, поговорим о требованиях. Что это такое, мы, в общем, понимаем - когда заказчик описывает нам, чего же именно он хочет, мы всегда слышим фразы типа "хотелось бы, чтобы проверка обновлений проводилась автоматически, как в антивирусах", "хочу большую зеленую кнопку в центре окна, которая начинает процесс", "*программа* должна позволять просматривать и печатать отчеты", "и чтоб красивенько все было, с полупрозрачностями, как в Висте", "при выходе должно выводиться подтверждение" и т. д. и т. п. Конечно, как настоящие разработчики, мы понимаем и то, что заказчик никогда не знает, что именно ему нужно, а если понимает, то объяснить не может. Но ведь фразы-то всегда, *по* сути, одинаковы! Они описывают, как заказчик представляет себе систему, чего заказчик хочет от системы, функциональность, которой он от нее ожидает, требования, которые к ней предъявляет.

Если обратиться к классикам, например, к той же "банде трех" (Якобсон, Буч, Рамбо), мы узнаем, что *требование - это желаемая функциональность, свойство или поведение системы*. Именно со сбора требований начинается процесс разработки *ПО*. Если изобразить процесс разработки *ПО* в виде " *черного ящика* " (уверены, читатель знает, что это такое, если нет - "Википедия" к вашим услугам), на выходе которого мы получаем *программный продукт*, то на *вход* этого "черного ящика" будет подаваться именно набор требований к программному продукту (рисунок 6.1)!

Рисунок 6.1.

Кстати, какую диаграмму напоминает этот рисунок? Правильно, *диаграмму активностей*. И выбор именно этой диаграммы тут абсолютно оправдан - помните, мы говорили, что *диаграммы активностей* часто используют для описания бизнес-процессов? Единственный нюанс: обычно процесс разработки не заканчивается с выпуском программного продукта - грядет новая *итерация*, новые, уточненные требования, новая версия и т. д.

Кстати, вернемся к требованиям. Да, мы сказали, что на вход нашего "черного ящика" подается набор требований. Но в какой форме? Как их документируют, эти требования? Думаю, большинство читателей помнит, что такое *техническое задание* - основной документ, без составления которого не начинался в советские времена ни один проект. Документ это был большой, многостраничный, с четкой структурой, определяемой ГОСТами (государственными отраслевыми стандартами). И описывал он, *по* сути, не что иное, как требования к создаваемой системе!

Техническое задание - вещь *по*-своему хорошая. Но время шло, менялись стандарты, нотации, способы описания требований. И вот постепенно *техническое задание* уступило *место* набору артефактов, состоящему из документов двух видов:

- диаграммы прецедентов;
- нефункциональные требования.

Диаграммы прецедентов составляют *модель прецедентов* (вариантов использования, use-cases). *Прецедент* - это функциональность системы, позволяющая пользователю получить некий значимый для него, ощутимый и измеримый результат. Каждый *прецедент* соответствует отдельному сервису, предоставляемому моделируемой системой в ответ на *запрос* пользователя, т. е. определяет способ использования этой системы. Именно *по* этой причине use cases, или прецеденты, часто в русской терминологии фигурируют как *варианты использования*. Варианты использования чаще всего применяются для спецификации внешних требований к проектируемой системе или для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно описывают типичные способы взаимодействия пользователя с системой, позволяющие корректно работать с предоставляемыми системой сервисами.

Нефункциональные требования - это описание таких свойств системы, как особенности среды и реализации, *производительность*, *расширяемость*, *надежность* и т. д. Часто нефункциональные требования не привязаны к конкретному варианту использования и потому выносятся в отдельный *список* дополнительных требований к системе (рисунок 6.2).

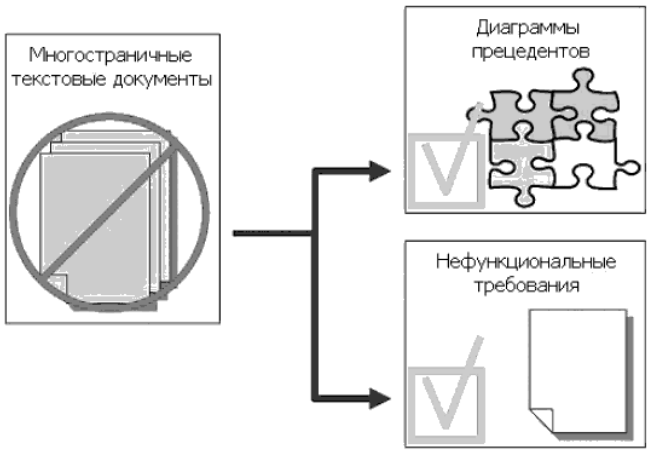


Рисунок 6.2.

Но вернемся же к прецедентам (вариантам использования). Идентифицировать прецеденты и действующие лица - обязанность системного аналитика. И делает он это для того, чтобы:

- четко разграничить систему и ее окружение;
- определить, какие действующие лица и как именно взаимодействуют с системой, какой функционал (варианты использования) ожидается от системы;

- определить и описать в словаре предметной области (глоссарии) общие понятия, которые необходимы для детального описания функционала системы (прецедентов).

Подобный вид деятельности обычно выполняется в такой последовательности:

1. Определение действующих лиц.
2. Определение прецедентов.
3. Составление описания каждого прецедента.
4. Описание модели прецедентов в целом (этот этап включает в себя создание словаря предметной области).

Вначале требования оформляются в виде обычного текстового документа, который создается или самим пользователем, или пользователем и разработчиком вместе. Далее требования оформляют в виде таблицы. В левую колонку помещают прецеденты, а в правую - действующих лиц, участвующих в прецеденте.

Рассмотрим пример. Секретарь размещает на сервере *меню* обеденных блюд на неделю. Сотрудники должны иметь возможность ознакомиться с *меню* и сделать заказ, выбрав блюда на каждый день следующей недели. *Офис*-менеджер должен иметь возможность сформировать счет и оплатить его. Система должна быть написана на *ASP.NET*. Такое вот нехитрое интернет-*приложение* для автоматизации заказов обедов в *офис*.

Думаем, здесь все понятно. *Таблица* с описанием требований может быть, например, такой:

Прецедент	Действующее лицо
разместить меню	секретарь
ознакомиться с меню	сотрудник, секретарь, офис-менеджер
сделать заказ	сотрудник, секретарь, офис-менеджер
сформировать счет	офис-менеджер
оплатить счет	офис-менеджер

Здесь нигде не сказано о том, что система должна быть написана на *ASP.NET*. Почему - понятно: это ведь нефункциональное требование! И еще, очевидно, что секретарь и *офис*-менеджер тоже являются сотрудниками. Читатель, внимательно прочитавший предыдущие материалы, заподозрит, что в данном случае, создавая модель прецедентов, говоря о действующих лицах, можно бы применить генерализацию. Действительно, *диаграмма прецедентов,* построенная на основе этой таблицы, может быть, например, такой (рисунок 6.3):

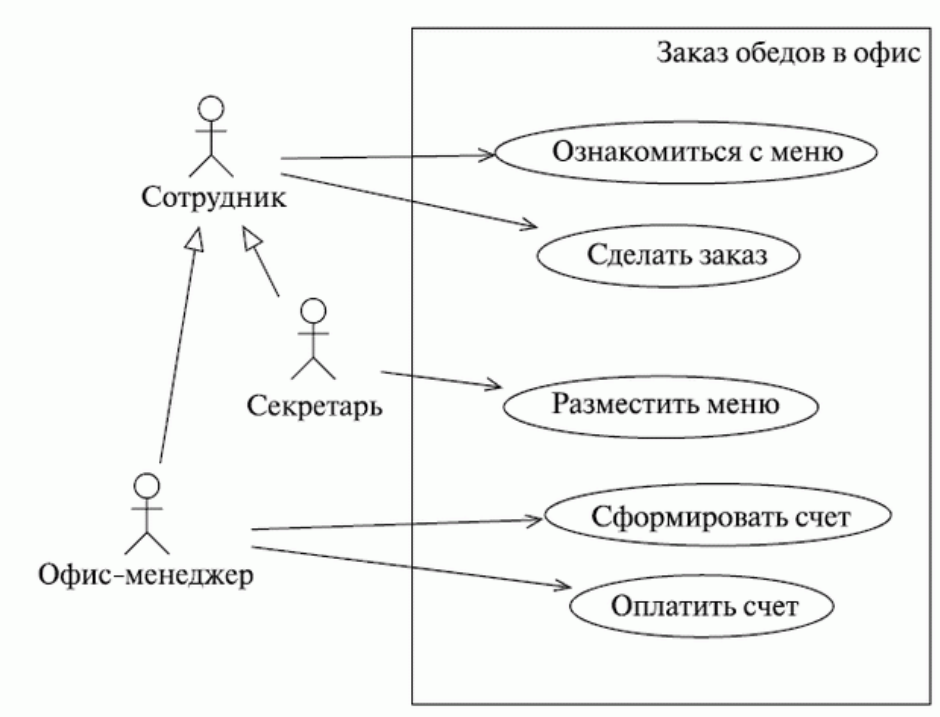


Рисунок 6.3.

Диаграммы прецедентов и их нотация

Что ж, у нас есть пример диаграммы. Итак, какие же элементы мы на ней видим? Первое, что бросается в глаза, - большой *прямоугольник*, внутри которого размещаются эллипсы, обозначающие, как мы уже поняли, прецеденты. В верхней части прямоугольника указано название моделируемой системы, а называют его *рамками системы* (*system boundary*, *subject boundary*), *контекстом* или просто *системой*. Этот элемент диаграммы показывает границу между тем, что вы как *аналитик* показали в виде прецедентов (внутри этих рамок), и тем, что вы изобразили как действующие лица (вне их). Чаще всего таким прямоугольником показывают *границы самой моделируемой системы*. То есть внутри границы находятся прецеденты - тот функционал, который реализует система (и в этом смысле прецеденты могут рассматриваться как представления подсистем и классов модели), а снаружи - *действующие лица*: пользователи и другие *внешние сущности*, взаимодействующие с моделируемой системой.

Следует сказать, что рамки системы на диаграммах прецедентов изображают довольно редко, т. к. они неявно подразумеваются самой диаграммой. *По* сути, этот элемент не привносит в диаграмму какой-либо дополнительной значимой информации, так что его использование - дело вкуса аналитика. Появление рамок системы на диаграмме прецедентов чаще всего диктуется особенностями персонального стиля проектирования.

Кроме рамок системы или ее контекста на диаграмме мы видим еще два вида связанных с ней сущностей - это *действующие лица* (экторы, actors) и *прецеденты*. Начнем с экторов. Довольно часто в русскоязычной литературе *по UML* для обозначения действующих лиц

можно встретить термин "*актер*". В принципе, смысл его более-менее понятен и оригинальному английскому термину он созвучен. Более того, есть еще одна причина такого перевода. Какое *слово* первым приходит к вам в голову, когда вы слышите *слово* "*актер*"? Да, конечно же - *слово* "роль"! Именно о ролях мы вскоре и поговорим, когда будем пытаться разобраться, что скрывается за понятием "действующее лицо". А пока, да простит нас читатель, далее мы все же будем пользоваться словом "эктор" - транскрипцией оригинального термина. Помнится, мы уже как-то писали о нашем отношении к буквальному переводу терминологии...

Итак, какой же смысл вкладывают в понятие эктора? *Эктор* - это набор ролей, которые исполняет *пользователь* в ходе взаимодействия с некоторой сущностью (системой, подсистемой, классом). Эктор может быть человеком, другой системой, подсистемой или классом, которые представляют нечто за пределами рассматриваемой сущности. Экторы "общаются" с системой путем обмена сообщениями. Четко выделив экторов, вы тем самым ясно определяете границу между тем, что внутри системы, и тем, что снаружи, - рамки системы.

Возможно, слова "роли, исполняемые пользователем" в определении эктора звучат не очень понятно. Очень забавно это понятие объясняется в Zicom Mentor:

роль - это не конкретный пользователь, а подобие шляпы, которую человек надевает, когда взаимодействует с сущностью.

"Физический" *пользователь* может играть роль одного или даже нескольких экторов, выполняя их функции в ходе взаимодействия с системой. И наоборот, роль одного и того же эктора может выполняться несколькими пользователями.

На диаграммах *UML* экторы изображаются в виде стилизованных человечков, ведь, как вы, конечно, помните, идея была в создании нотации, любой символ которой легко может быть изображен от руки (рисунок 6.4):

Рисунок 6.4.

Несмотря на "человеческий" вид этого обозначения, не следует забывать, что экторы - это не обязательно люди. Эктором, как мы уже говорили ранее, может быть внешняя система, подсистема, *класс* и т. д. Кстати, человечек ("stick-person") - это не единственное обозначение эктора, используемое в *UML*. На диаграммах прецедентов обычно применяется именно "человекоподобная" форма эктора, но на других диаграммах, и особенно *в случаях,*

когда эктор имеет атрибуты, которые важно показать, используется изображение эктора как класса со стереотипом <<actor>> (рисунок 6.5):

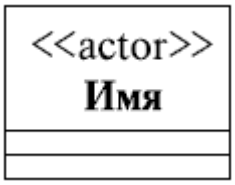


Рисунок 6.5.

С системой экторы, как мы уже сказали, общаются через сообщения, но если говорить на более высоком уровне абстракции, в терминах модели прецедентов, то взаимодействуют они с системой через прецеденты. Один и тот же эктор может быть связан с несколькими прецедентами, и наоборот, один *прецедент* может быть связан с несколькими разными экторами. Ассоциации между эктором и прецедентом всегда бинарные - т. е. представляют отношения типа "один к одному", использование кратности недопустимо. Это не противоречит сказанному выше: действительно, один эктор может быть связан с несколькими прецедентами, но только с помощью отдельных ассоциаций - *по* одной на каждый *прецедент*. Мы видели это в нашем примере. Кстати, там мы видели ассоциации, изображенные не просто в виде линий, а стрелками. Думаем, смысл этого обозначения вполне понятен: это *направленная ассоциация* и стрелка (как и на других диаграммах) всегда направлена в сторону той сущности, от которой что-то требуют, чьим сервисом пользуются и т. д.

И еще - экторы не могут быть связаны друг с другом. Единственное допустимое *отношение* между экторами - *генерализация* (*наследование*). Опять-таки, в нашем примере с заказом обедов в *офис*, вы могли увидеть именно такой вид отношений между экторами. Это не значит, что в реальной жизни *офис*-менеджер и секретарь (да и вообще любые два сотрудника) не могут общаться: просто при создании модели прецедентов такое общение не попадает в область наших интересов, считается несущественным.

Еще один тип элементов, встречающийся на диаграммах прецедентов, более того, давший им название, - это собственно *прецеденты*, или варианты использования. *Прецедент* - это описание набора последовательных событий (включая возможные варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецеденты описывают сервисы, предоставляемые системой экторам, с которыми она взаимодействует. Причем *прецедент* никогда не объясняет, "как" работает сервис, а только описывает, "что" делается.

Изображаются прецеденты в виде эллипса, внутрь контура которого помещается имя (описание) прецедента. Имя прецедента обычно намного длиннее имен других элементов мо-

дели. Почему это так, в принципе, понятно: имя прецедента описывает взаимодействие эктора с системой, говорит о том, какими сообщениями они обмениваются между собой. В нашем примере с заказом обедов мы видели несколько прецедентов и наверняка читатель заметил, что имя прецедента - это, скорее, название сценария, воспроизводящегося в ходе взаимодействия эктора с системой. Причем это всегда описание *с точки зрения эктора*, описание услуг, предоставляемых системой пользователю. Приведем пример простейшей диаграммы, иллюстрирующей сказанное нами об обозначениях прецедента (рисунок 6.6).

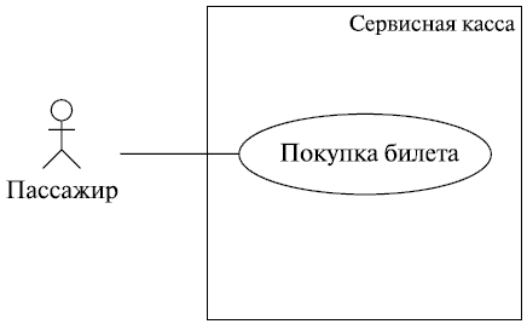


Рисунок 6.6.

В этом примере пассажир может купить в сервисной кассе билет на некоторый вид транспорта. Покупка билета - это название сценария, *по* которому эктор (пассажир) может взаимодействовать с системой (кассой). Заметьте, это *не описание* сценария, а именно название - оно говорит нам, *что* делает эктор в процессе взаимодействия, но не говорит, как именно! И еще - прецеденты определяют *непересекающиеся* сценарии поведения. Выполнение одного прецедента не может быть прервано в результате работы другого прецедента. Другими словами, выполнение одного прецедента не может быть прервано в результате событий или действий, вызванных выполнением другого прецедента. Прецеденты выступают как *атомарные транзакции*, выполнение которых не может быть прервано.

Внимательный читатель, возможно, отметил то, как незаметно мы ввели в употребление *слово* " **сценарий** ". Что же такое *сценарий* и как понятие сценария связано с понятием прецедента? На первый вопрос хорошо отвечают классики (Г. Буч):

Сценарий - это конкретная последовательность действий, иллюстрирующая поведение.

Сценарий - это повествовательный рассказ о совершаемых эктором действиях, история, эпизод, происходящий в данных временных рамках и данном контексте взаимодействия. Сценарии (в различных формах представления) широко применяются в процессе разработки программного обеспечения. Как мы уже только что отметили, написание сценария напоминает написание художественного рассказа, и этим объясняется тот факт, что использование сценариев широко распространено среди аналитиков, которые часто обладают художествен-

ными или литературными способностями. Несмотря на непрерывный повествовательный характер, сценарии можно рассматривать как последовательности действий (делать *раскадровку*). При разработке пользовательского интерфейса сценарии описывают взаимодействие между пользователем (или категорией пользователей, например, администраторами системы, конечными пользователями) и системой.

Такой *сценарий* состоит из последовательного описания комбинаций отдельных действий и задач (например, нажатий клавиш, щелчков *по* элементам управления, ввода данных в соответствующие поля и т. д.). Вспомните, к примеру, описания последовательностей действий пользователя (предназначенных для достижения определенных результатов, решения определенных задач), которые вы находите в справке к малознакомой программе. То же самое можно сказать о модных сейчас "how-to videos", в которых такие последовательности отображаются визуально, на конкретных примерах. В любом случае, цель подобных справочных материалов - предоставить описание типичных сценариев использования системы, сценариев взаимодействия между пользователем и системой.

Сценарии также иногда можно увидеть на диаграмме прецедентов. Иногда их изображают в виде " *листа бумаги* ", на котором написано *имя файла*, - прямоугольника с загнутым нижним левым уголком. В этом случае указанный *файл* содержит в себе описание данного сценария. А иногда *сценарий* записывается в комментарий. Как вы, наверное, помните, комментарии (ноутсы, notes) изображаются прямоугольниками с загнутым верхним правым углом и соединяются с элементом, который они поясняют, пунктирной линией (рисунок 6.7).

Рисунок 6.7.

Как мы уже упоминали, сценарии могут быть записаны в различных формах. Это может быть структурированный, но неформализованный текст, формализованный структурированный текст, *псевдокод*, *таблица*, *диаграмма активностей*, наконец! Каждый *сценарий* описывает в повествовательной форме завершенное, конкретное взаимодействие, имеющее с точки зрения пользователя определенную цель. Если рассматривать табличную форму представления сценария, то линия, разделяющая левый и правый столбцы таблицы, символизируют собой границу, отделяющую действия пользователя от ответных действий системы. Табличная форма особо подчеркивает участие пользователя, что является очень важным аспектом при разработке пользовательского интерфейса.

Вот пример простого (неформализованного) текстового описания сценария.

Пользователь вводит логин, пароль, адрес электронной почты и код подтверждения и нажимает кнопку "Далее". Система запрашивает ввод проверочного кода. Пользователь вводит код и нажимает кнопку "Далее". Система проверяет соответствие кода изображенному на картинке.

Не правда ли, знакомая процедура? Да, это описание регистрации пользователя на некотором сайте. Правда, не совсем полное: не рассмотрены случаи, когда выбранный пользователем логин уже занят, *адрес* электронной почты введен неправильно, *пароль* не удовлетворяет требованиям или код не соответствует изображенному на картинке. О таких случаях - *альтернативных сценариях* - мы поговорим чуть позже.

А вот тот же *сценарий* в табличном представлении:

Действия пользователя	Реакция системы
Ввод логина, пароля, адреса электронной почты и нажатие кнопки "Далее"	Запрос ввода проверочного кода
Ввод проверочного кода и нажатие кнопки "Далее"	Проверка кода на соответствие изображенному на картинке

Вы, конечно, заметили, что этот *сценарий* можно детализировать - например, прежде чем попросить ввести проверочный код, система отображает картинку, на которой этот самый код изображен. Т. е. *запрос* на ввод кода *включает в себя вывод* картинки с упомянутым кодом. Об этом мы тоже еще поговорим.

А пока попробуем ответить на второй вопрос, а именно: *как связаны понятия сценария и прецедента*. Прецеденты, как мы уже говорили, рождаются из требований к системе. Но говорят они о том, что делает система. Как система это делает, говорят сценарии. Таким образом, *прецедент* можно специфицировать путем описания потока действий или событий в текстовой форме - в виде, понятном для "постороннего" (не занятого в непосредственной разработке системы) читателя. А ведь такое описание - это и есть *сценарий*! Таким образом, *сценарии специфицируют прецеденты*. И еще. Поскольку сценарии - это, *по* сути, рассказы, они являются весьма эффективным средством извлечения информации из бесед с заказчиком и предоставляют превосходное, понятное непрофессионалу описание создаваемого приложения. Сценарии, да и вообще *диаграммы прецедентов* (дополненные сценариями) являются отличным *средством общения между разработчиками и заказчиком*, причем, в силу простоты нотации, - средством, понятным обеим сторонам. В конечном итоге, взаимосвязь между требованиями, прецедентами и сценариями можно изобразить такой "псевдодиаграммой" (рисунок 6.8).

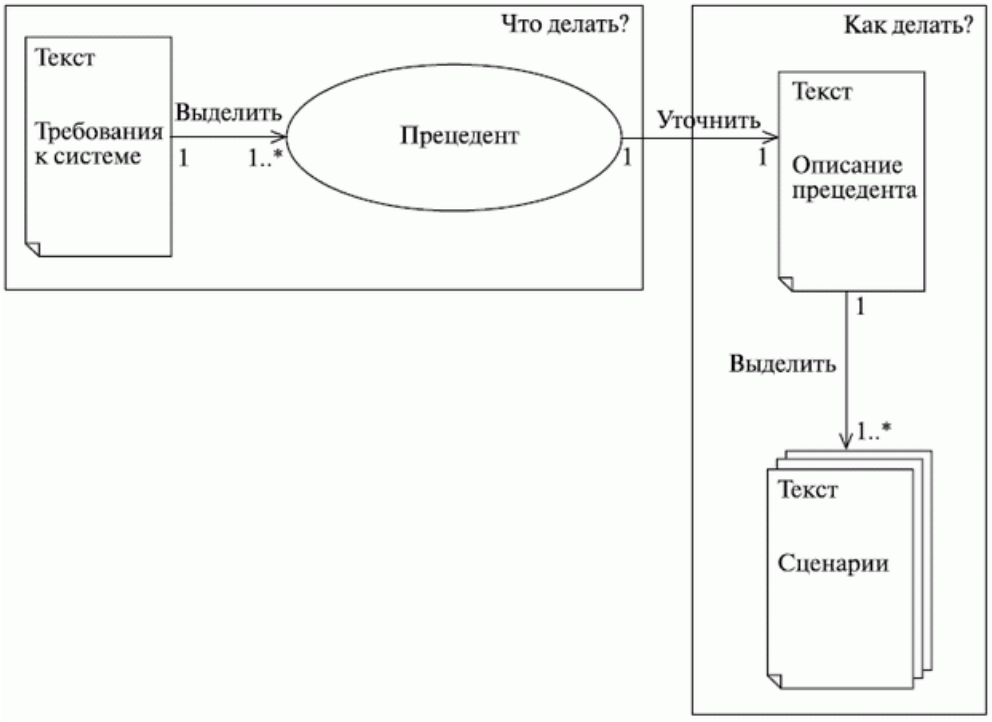


Рисунок 6.8.

Как видите, для каждой ассоциации на диаграмме проставлена *кратность* и ее смысл вполне понятен, но все же о кратности следует поговорить отдельно. Один *прецедент* определяет несколько сценариев, каждый из которых представляет один из возможных вариантов определяемого прецедентом потока событий. Сценарии так же соотносятся с прецедентами, как экземпляры класса, т.е. *сценарий - это экземпляр прецедента*, как *объект* - экземпляр класса. Система может содержать, например, несколько десятков прецедентов, каждый из которых, в свою *очередь*, может разворачиваться в десятки сценариев. Как правило, *прецедент* описывает не одну последовательность действий, а множество, и выразить все детали рассматриваемого прецедента с помощью одной последовательности действий обычно не получается. Практически для любого прецедента можно выделить основной *сценарий*, описывающий "нормальную" последовательность действия, и *вспомогательные*, описывающие *альтернативные* последовательности, которые инициируются в случае возникновения определенных условий.

Другой вопрос: требуется ли такое уточнение модели прецедентов, оправдано ли оно для данного уровня приближения, или "подразумевающиеся" *альтернативные* сценарии можно опустить? Например, в предыдущем примере с покупкой билета в сервисной кассе мы не изобразили сценарии (и, соответственно, прецеденты), соответствующие вариантам, когда билетов на выбранный пассажиром рейс уже не осталось, пассажир изменил свое решение и хочет взять билет на другой рейс, когда *оплата* идет наличными или *по* кредитной карте и т. д.

"Хватит ходить вокруг да около!" - воскликнет нетерпеливый читатель. Уже заканчиваем. Мы просто хотели мягко подвести читателя к вопросу об отношениях между прецедентами. А отношения эти весьма многообразны. Начнем со старого знакомого - отношения обобщения (наследования, генерализации). О генерализации мы уже говорили не раз, когда рассматривали *диаграммы классов*. Но все же напомним суть этого понятия. Как говорят классики, *обобщение* - это *отношение* специализации (обобщения), в котором объекты специализированного элемента (потомка) могут быть подставлены вместо объектов обобщенного элемента (родителя, или предка).

Точно так же, как мы обычно поступаем с классами, после того как мы выделили и описали каждый *прецедент*, мы должны просмотреть их все на предмет наличия одинаковых действий - поискать, а не выполняются ли (используются) некоторые действия совместно несколькими вариантами использования. Этот совместно используемый фрагмент лучше описать в отдельном прецеденте. Таким образом мы уменьшим *избыточность* модели за счет применения обобщения прецедентов (иногда, правда, говорят не об обобщении, а об *использовании* прецедентов; почему - сейчас поймете). Как это и "положено" при наследовании, экземпляры обобщенных прецедентов (потомков) сохраняют поведение, присущее обобщающему прецеденту (предку). Другими словами, наличие (использование) в варианте использования `X` обобщенного варианта использования `Y` говорит нам о том, что экземпляр прецедента `X` *включает в себя* поведение прецедента `Y`. Обобщения применяются, чтобы упростить понимание модели вариантов использования за счет многократного задействования "заготовок" прецедентов для создания прецедентов, необходимых заказчику (помните, как мы рассматривали вопрос о том, всегда ли необходимо создавать новый *класс*, или лучше воспользоваться готовым решением, чувствуете аналогию?). Такие "полные" прецеденты называются *конкретными прецедентами*. "Заготовки" прецедентов, созданные лишь для многократного использования в других прецедентах, называют абстрактными прецедентами. Абстрактный *прецедент* (как и *абстрактный класс*) не существует сам *по* себе, но экземпляр конкретного прецедента демонстрирует поведение, описываемое абстрактными прецедентами, которые он (повторно) использует. *Прецедент*, который экторы наблюдают при взаимодействии с системой ("полный" *прецедент*, как мы называли его ранее), часто называют еще " *реальным* " прецедентом.

Как мы уже говорили выше, *обобщение* (*наследование*) чаще всего используют между классами и интерфейсами. Однако другие элементы модели также могут находиться между собой в отношении наследования - например, пакеты (о которых мы тут не говорим), экторы, прецеденты...

Изображается *обобщение*, как, конечно, помнит внимательный читатель, линией с "незакрашенной" треугольной стрелкой на конце. *Обобщение* - это *отношение* между предком и потомком, и стрелка всегда указывает на предка. Если вспомнить, что потомки наследуют (используют) свойства предка, то вполне логично вспоминается наше утверждение о том, что стрелки в *UML* всегда направлены в сторону того, от кого что-то требуют, чьими сервисами пользуются (рисунок 6.9):

Рисунок 6.9.

Как мы уже говорили ранее и видели в нашем первом примере *диаграммы прецедентов, обобщение* может использоваться для создания различных разновидностей экторов. Экторы-потомки наследуют от предка базовые характеристики и дополняют их своей спецификой. Точно так же *прецедент-потомок* наследует поведение и семантику прецедента-родителя и дополняет его поведение.

Следующий вид отношений между прецедентами - включение. *Отношение* включения означает, что *в некоторой точке базового прецедента содержится поведение другого прецедента*. Включаемый *прецедент* не существует сам *по* себе, а является всего лишь частью объемлющего прецедента. Таким образом, базовый *прецедент* как бы заимствует поведение включаемых, раскладываясь на более простые прецеденты. Например, когда мы покупаем в магазине некоторую вещь, в момент считывания кассиром штрих-кода обновляется состояние *базы данных* товаров, имеющихся в наличии, - количество наличных единиц купленного товара уменьшается. То же самое действие выполняется и в том случае, если купленный *товар* оказался бракованным, непригодным к использованию или попросту нам не понравился: состояние упомянутой *базы данных* вновь обновляется - но теперь уже в сторону увеличения количества наличных единиц определенного товара. Т. е. оба этих действия - и покупка, и возврат - содержат (включают в себя) такое действие, как обновление содержимого *БД*.

А как же изображается включение? Да очень просто - как зависимость (пунктирная линия со стрелкой, помните?) со стереотипом <<include>>. При этом стрелка направлена, естественно, в сторону включаемого прецедента. Этот факт легко объяснить, если вспомнить утверждение, которое мы уже несколько раз использовали в этом курсе: стрелка всегда направлена в сторону того элемента, от которого что-то требуется, чьими сервисами пользуются. А если считать, что объемлющий *прецедент* включает в себя, заимствует (использует) поведение включаемых прецедентов, становится ясно, что стрелка может быть направлена только таким образом. А вот и *диаграмма*, иллюстрирующая вышесказанное, которую мы позаимствовали из Zicom Mentor (рисунок 6.10).

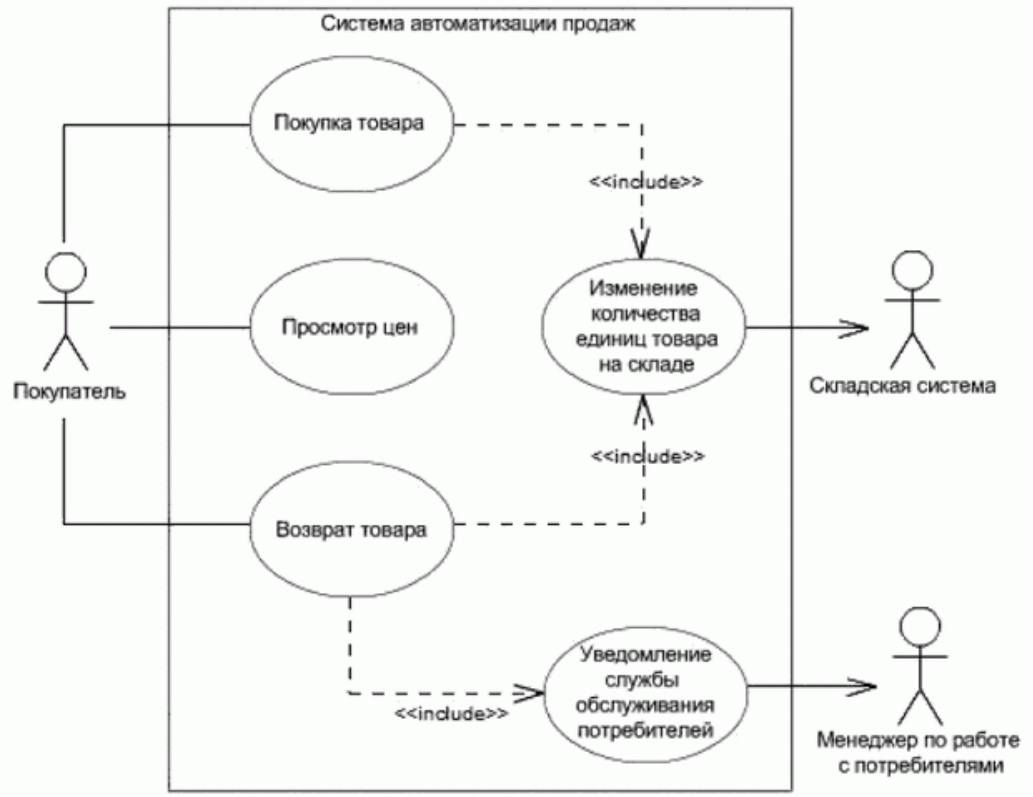


Рисунок 6.10.

Как хорошо видно из этого примера, использование включения позволяет избежать многократного описания одного и того же набора действий - общее поведение можно просто описать в виде прецедента, включаемого в базовые.

На очереди - *отношение расширения*. Чтобы уяснить себе смысл расширения, представим себе, что мы говорим об оплате некоторого купленного нами товара. Мы можем оплатить *товар* наличными, если сумма не превышает $ 100. Или оплатить кредитной картой, если сумма находится в пределах от $ 100 до $ 1000. Если же сумма превышает $ 1000, нам придется брать *кредит*. Таким образом мы расширили понимание *операции* оплаты купленного товара и на случаи, когда используются другие средства оплаты, нежели наличные. Но сами эти случаи возникают только при строго определенных условиях: когда *цена товара* попадает в определенные рамки.

Расширение дополняет *прецедент* другими прецедентами, "срабатывающими" при некоторых условиях, - просто добавляет в исходный *прецедент* последовательность действий, содержащуюся в другом прецеденте. *Отношение* расширения прецедента А к прецеденту В означает, что экземпляр прецедента В может включать в себя (при определенных условиях, которые могут быть описаны в расширении; как именно описаны, мы скажем чуть позже) поведение, описанное в прецеденте А. Пример показан на следующей диаграмме (рисунок 6.11):

Рисунок 6.11.

Однако в приведенном примере не видно, при каких именно условиях человек использует каждый конкретный способ оплаты. В то же время, при моделировании с использованием расширения можно указать как условия осуществления расширенного поведения, так и *место - точку расширения* прецедента, в которой подключаются действия из расширяющих прецедентов. Вспомните *оператор безусловного перехода*, который вы, надеемся, использовали в своих программах не слишком часто. Как только *интерпретатор* доходит до этого оператора, он передает управление на строку, которая помечена меткой, указанной в этом операторе. Правда, в случае расширения речь идет скорее об операторе условного перехода - когда исходный *прецедент* (а именно, последовательность действий, содержащаяся в нем) приходит в точку расширения, происходит оценка условий расширения. Если условия выполняются, *прецедент* включает в себя последовательность действий из расширяющего прецедента.

Точка расширения описывается в *дополнительном разделе* прецедента, отделенном от его названия горизонтальной линией - точно так же, как в отдельных разделах перечисляются атрибуты класса и его *операции*. Ниже показан пример описания точки расширения, позаимствованный нами из Zicom Mentor (рисунок 6.12).

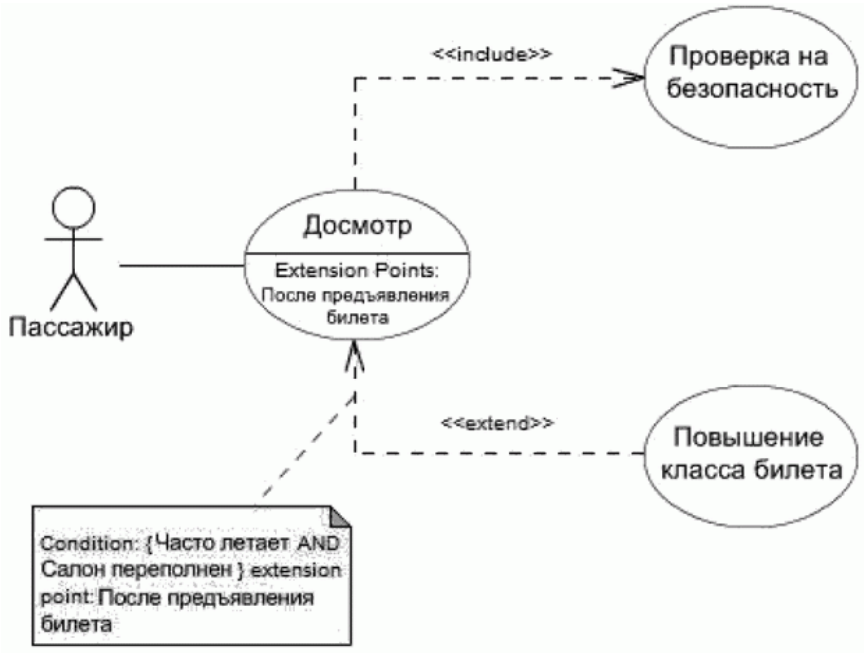


Рисунок 6.12.

В этом примере *регистрация* пассажиров авиарейса включает в себя *контроль* службы безопасности, а при условии (указанном в примечании после служебного слова "*Condition*:"), что человек часто летает и салон переполнен (обратите внимание на оператор *AND*, говорящий об одновременности выполнения условий), *класс* билета может быть повышен, например, с "эконом" до "бизнес-класса". Причем такой апгрейд может произойти только после того, как билет предъявлен на стойку регистрации - это и есть точка расширения. Она описана (ее имя указано) в *дополнительном разделе* прецедента после служебной фразы "*Extension* points:". Предваряя вопрос читателя, скажем, что *прецедент* может иметь сколь угодно много точек расширения. А сопоставить конкретный расширяющий *прецедент* с определенной точкой расширения можно, прочитав условия расширения, указанные в комментариях, - само условие записывается после служебного слова "*Condition*:" в фигурных скобках, за которыми идет служебная фраза "*Extension point*:", и после нее указывается имя точки расширения. Посмотрите еще раз на наш пример с регистрацией пассажиров в аэропорту и убедитесь сами, что все это очень просто!

Некоторое недоумение может вызвать то, что стрелка направлена всегда в сторону расширяемого прецедента. Но и это легко объяснить с точки зрения нашего тезиса, что "стрелка всегда указывает на того, от которого что-то требуют": ведь для того, чтобы *прецедент* был расширен, нужно, чтобы он попал в точку расширения и проверилась истинность условий - только тогда действия, содержащиеся в расширяющем прецеденте, смогут быть добавлены в последовательность действий исходного прецедента. Так что все правильно - от расширяемого прецедента требуется точка расширения и проверка условий, потому и стрелка направлена к нему.

Подытоживая все вышесказанное, можно сказать, что *расширение позволяет моделировать необязательное поведение системы* (был бы *класс* билета повышен, если бы пассажир не налетал нужного количества миль, а салон был бы почти пуст?). Сам факт расширения зависит от выполнения условий - расширения ведь может и не произойти! Это просто отдельные последовательности действий, выполняемые лишь при определенных обстоятельствах и включаемые в определенных точках сценария (обычно в результате явного взаимодействия с эктором).

Организация прецедентов с помощью выделения общего поведения (включение) и различных вариантов поведения (расширение) - важная составляющая часть процесса разработки простого, сбалансированного и понятного набора прецедентов. Можно сказать, даже, что использование включения и расширения - признак хорошего стиля в моделировании прецедентов.

На этом разговор о нотации диаграмм прецедентов можно было бы и завершить. Хотелось бы только сказать еще пару слов о соотношении между понятиями прецедента и *кооперации*. О кооперации мы уже говорили ранее (помните *диаграммы взаимодействия*?) как о множестве ролей, работающих вместе, чтобы обеспечить некоторое поведение системы. Мы также упоминали о том, что прецеденты отвечают на вопрос "что делает система?", но не говорят, как именно она это делает. На этапе анализа понимать, как именно система реализует свое поведение, действительно не нужно. Но при переходе к реализации неплохо бы знать, *какие именно классы (или другие элементы модели), совместно работая, обеспечивают нужное поведение*. То есть мы логично перешли от разговора о прецедентах к разговору о кооперации! Недаром обозначения кооперации и прецедента очень похожи (читатель, конечно, помнит, что *кооперация* обозначается пунктирным эллипсом) (рисунок 6.13).

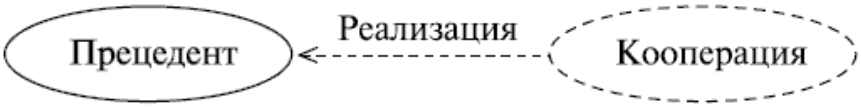


Рисунок 6.13.

Так в каком же отношении находятся *прецедент* и *кооперация*? Из предыдущего абзаца логично следует, что это *отношение* реализации. Каждый *прецедент* реализуется одной или несколькими кооперациями. Это, конечно, не означает, что классы жестко распределены *по* кооперациям: классы, принимающие участие в кооперации, реализующей определенный *прецедент*, будут участвовать и в других кооперациях.

Моделирование при помощи диаграмм прецедентов

Модель прецедентов, *по* сути, является концептуальной моделью системы. В ней, как мы уже не раз отмечали, в общих чертах описывается только поведение (функциональность)

системы, а о деталях реализации речь не идет - на данном этапе реализация не важна, гораздо важнее собрать требования к системе и оформить их в наглядном виде, понятном и разработчикам, и заказчику.

Итак, подводя итоги, мы можем сформулировать три причины использования прецедентов. Или, вернее, три способа использования прецедентов (не случайно в русском переводе частенько можно встретить *словосочетание* "*вариант использования*"!) в ходе работы над системой:

1. *Прецеденты дают возможность аналитикам, пользователям и разработчикам говорить на одном языке*: используя прецеденты, аналитики (эксперты в предметной области) могут на основе пожеланий заказчика описать поведение системы с точки зрения пользователя с такой степенью детализации, что разработчики смогут без труда сконструировать "внутренности" системы. В то же время, нотация диаграмм прецедентов настолько проста, что даже неподготовленный пользователь (заказчик) способен понять их смысл и помочь в их уточнении - ведь картинки (а тем более комиксы, каковыми, по сути, являются диаграммы UML) воспринимаются намного легче, чем текст!

2. *Прецеденты позволяют разработчикам понять назначение элемента*: система, подсистема или даже класс могут быть сложными образованиями, состоящими из большого числа составных частей и имеющими большое число атрибутов и операций. Моделирование прецедентов позволяет лучше представить себе поведение системы, понять, какие элементы модели играют какие роли в реализации этого поведения, в какие кооперации входят, и какой именно прецедент (функционал системы) реализуют.

3. *Прецеденты являются основой для тестирования элемента в течение всей разработки*: модель прецедентов описывает желаемое поведение системы (ее функционал) с точки зрения пользователя. Так что, постоянно сопоставляя предоставляемый элементом (фактический) функционал с имеющимися прецедентами, можно надежно контролировать корректность реализации элемента. Вот вам и надежный источник регрессионных тестов. Кроме этого, появление нового прецедента зачастую заставляет пересмотреть реализацию элемента, дабы убедиться, что она обладает достаточной гибкостью, изменяемостью и масштабируемостью.

Прецеденты полезны и для прямого, и для обратного проектирования. При прямом проектировании мы, *по* сути, осуществляем "перевод" с *UML* на некий *язык программирования*. И тестировать созданное *приложение* следует, основываясь именно на потоках событий, описываемых прецедентами. *Обратное проектирование* предполагает перевод с языка программирования на язык *UML*-диаграмм. Такими вещами приходится заниматься в силу ряда причин:

1. С целью поиска ошибок и чтобы убедиться в адекватности дизайна: отличная идея после первого перевода с UML на язык программирования сделать обратный перевод и сравнить исходные и восстановленные UML-модели (желательно, чтобы эти переводы выполнялись разными командами). Это позволит убедиться в том, что дизайн системы соответствует модели, никакая информация в ходе перевода не была утеряна, да и попросту выловить некоторые "баги". Такой подход называется обратной семантической трассировкой (или RST - Reverse Semantic *Traceability*) и разрабатывается компанией INTSPEI (http://www.intspei.com) как одна из базовых техник методологии INTSPEI P-Modeling Framework, краткие сведения о которой вы можете найти в приложении к этому курсу.

2. Когда *отсутствует документация*: иногда стоит задача модификации существующей системы, код которой плохо документирован. В таком случае перевод с языка программирования на язык UML-диаграмм - отличный способ понять назначение системы и ее частей, функционал, предоставляемый ею, и т. д.

И наконец, следует отметить, что, конечно, только диаграмм прецедентов, как и сценариев, ими определяемых, недостаточно, чтобы создать модель поведения системы. Как мы уже не раз упоминали, прецеденты говорят, что делает система, но не говорят, как. Об этом говорят сценарии, но в текстовой форме, что делает их довольно сложными для восприятия. На помощь приходят диаграммы *взаимодействий, которые визуализируют сценарии*. Таким образом, мы теперь можем дополнить нашу старую "псевдодиаграмму" и на этом успокоиться (рисунок 6.14):

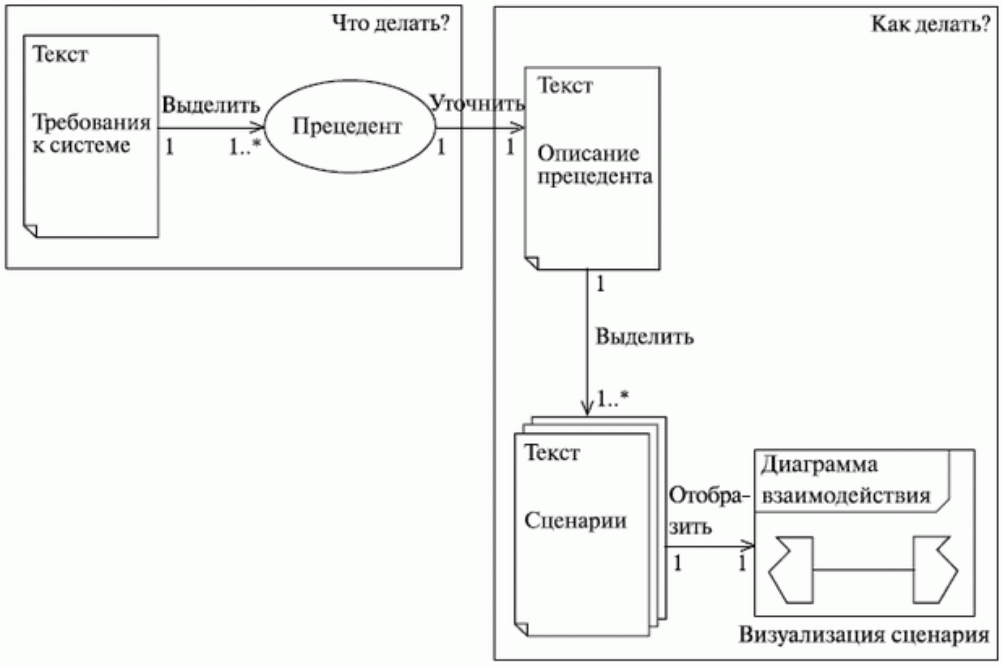


Рисунок 6.14.

В заключение приведем пару примеров законченных диаграмм прецедентов. Первый пример (смысл которого понятен и без дополнительных пояснений) демонстрирует включение, расширение и *наследование* прецедентов. Обратите внимание на стрелки, которые направлены к экторам, изображающим шлюзы. Все правильно - ведь система пользуется их услугами при отправке сообщений, в то время как маркетолог, наоборот, пользуется услугами системы, и потому стрелки направлены от него (рисунок 6.15).

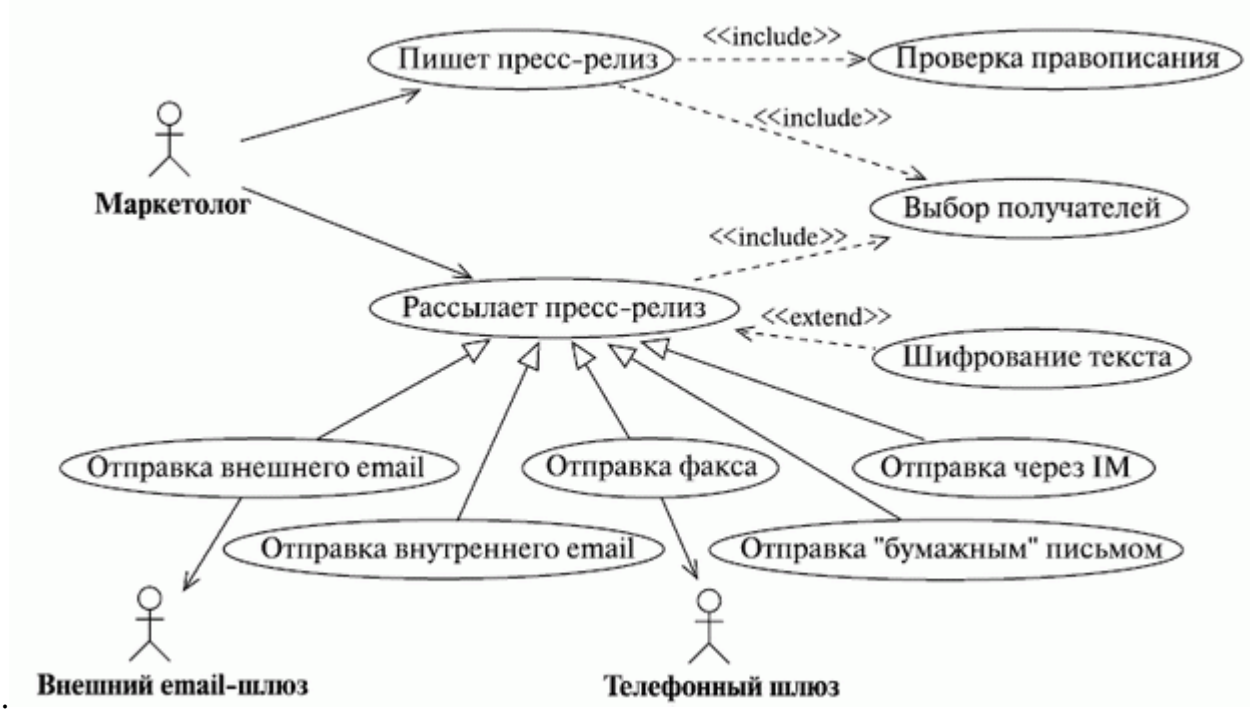


Рисунок 6.15.

Выводы

1. Модель прецедентов позволяет описать систему на концептуальном уровне.
2. *Диаграммы прецедентов* - отличное средство коммуникаций между экспертами, пользователями и разработчиками, а также основа для тестирования создаваемой системы.
3. Прецедент - это описание набора последовательных событий (включая возможные варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату.
4. Эктор - это набор ролей, которые исполняет пользователь в ходе взаимодействия с некоторой сущностью.
5. Прецеденты (как и экторы) могут быть генерализованы, т. е. наследовать и дополнять свойства своих предков.
6. Прецеденты также могут вступать между собой в отношения включения и расширения, что позволяет разложить прецеденты на более простые составляющие и выделить необязательное поведение.
7. Каждый прецедент реализуется одной или несколькими кооперациями.

8. Сценарии специфицируют прецеденты, а диаграммы взаимодействий визуализируют сценарии.

Контрольные вопросы

1. Что такое нефункциональные требования? Как они отображаются на диаграммах прецедентов?
2. Какие способы изображения экторов вы знаете?
3. В какие отношения могут вступать экторы между собой?
4. В чем состоит смысл отношений включения и расширения?
5. Что такое точка расширения?
6. Перечислите известные вам причины использования прецедентов.
7. Как прецеденты применяют в прямом и обратном проектировании?