МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра программной инженерии

УТВЕРЖДАЮ
Проректор по учебной работе
_____О.Г.Локтионова
«___» _____2017 г.

Объектно-ориентированный подход к моделированию предметной области

Методические указания к лабораторным работам
по курсу «Языки объектно-ориентированного
программирования»

Курск 2017

УДК 681.3

Составитель Е.И.Аникина

Рецензент
Кандидат технических наук, доцент кафедры программной инженерии *Н.Н. Бочанова*

Объектно-ориентированный подход к моделированию предметной области: методические указания к лабораторным работам по курсу «Языки объектно-ориентированного программирования»/Юго-Зап. гос. ун-т; сост. Е.И.Аникина. Курск, 2017. 38 с.

Содержит теоретические сведения и задания для выполнения лабораторных работ по изучению объектно-ориентированного подхода к моделированию предметной области информационных и прграммных систем на языке UML.

Предназначено для студентов направлений подготовки, входящих в группы «Компьютерные и информационные науки» и «Информатика и вычислительная техника».

Текст печатается в авторской редакции.

Подписано в печать . Формат 60x84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ .
Бесплатно.

Юго-Западный государственный университет
305040, Курск, ул.50 лет Октября, 94.

ВВЕДЕНИЕ

Компания, занимающаяся производством программного обеспечения, может преуспевать только в том случае, если выпускаемая ею продукция всегда отличается высоким качеством и разработана в соответствии с запросами пользователей. К сожалению, во многих организациях путают понятия "вторичный" и "несущественный". Нельзя забывать, что для разработки эффективной программы, которая соответствует своему предполагаемому назначению, необходимо постоянно встречаться и работать с пользователями, чтобы выяснить реальные требования к вашей системе. Если вы хотите создать качественное программное обеспечение, вам необходимо разработать прочное архитектурное основание проекта, открытое к возможным усовершенствованиям. Для быстрой и эффективной разработки программного продукта с минимальным браком требуется привлечь рабочую силу, выбрать правильные инструменты и определить верное направление работы. Чтобы справиться с поставленной задачей, принимая во внимание затраты на обеспечение жизненного цикла системы, необходимо, чтобы процесс разработки приложения был тщательно продуман и мог быть адаптирован к изменяющимся потребностям вашего бизнеса и технологии.

Центральным элементом деятельности, ведущей к созданию первоклассного программного обеспечения, является моделирование. Модели позволяют нам наглядно продемонстрировать желаемую структуру и поведение системы. Они также необходимы для визуализации и управления ее архитектурой. Модели помогают добиться лучшего понимания создаваемой нами системы, что зачастую приводит к ее упрощению и возможности повторного использования. Наконец, модели нужны для минимизации риска.

Объектное моделирование

Инженеры-строители создают огромное количество моделей. Чаще всего это *структурные* модели, позволяющие визуализировать и специфицировать части системы и то, как они соотносятся друг с другом. Иногда, в особо критичных случаях, создаются также *динамические* модели - например, если надо изучить поведение конструкции при землетрясении. Эти два типа различаются по организации и по тому, на что в первую очередь обращается внимание при проектировании.

При разработке программного обеспечения тоже существует несколько подходов к моделированию. Важнейшие из них - алгоритмический и объектно-ориентированный.

Алгоритмический метод представляет традиционный подход к созданию программного обеспечения. Основным строительным блоком является процедура или функция, а внимание уделяется прежде всего вопросам передачи управления и декомпозиции больших алгоритмов на меньшие. Ничего плохого в этом нет, если не считать того, что системы не слишком легко адаптируются. При изменении требований или увеличении размера приложения (что происходит нередко) сопровождать их становится сложнее.

Наиболее современным подходом к разработке программного обеспечения является *объектно-ориентированный*. Здесь в качестве основного строительного блока выступает объект или класс. В самом общем смысле объект - это сущность, обычно извлекаемая из словаря предметной области или решения, а класс является описанием множества однотипных объектов. Каждый объект обладает идентичностью (его можно поименовать или как-то по-другому отличить от прочих объектов), состоянием (обычно с объектом бывают связаны некоторые данные) и поведением (с ним можно что-то делать или он сам может что-то делать с другими объектами).

В качестве примера давайте рассмотрим простую трехуровневую архитектуру биллинговой системы, состоящую

из интерфейса пользователя, программного обеспечения промежуточного слоя и базы данных. Интерфейс содержит конкретные объекты - кнопки, меню и диалоговые окна. База данных также состоит из конкретных объектов, а именно таблиц, представляющих сущности предметной области: клиентов, продукты и заказы. Программы промежуточного слоя включают такие объекты, как транзакции и бизнес-правила, а также более абстрактные представления сущностей предметной области (клиентов, продуктов и заказов).

Объектно-ориентированный подход к разработке программного обеспечения является сейчас преобладающим просто потому, что он продемонстрировал свою полезность при построении систем в самых разных областях любого размера и сложности. Кроме того, большинство современных языков программирования, инструментальных средств и операционных систем являются в той или иной мере объектно-ориентированными, а это дает веские основания судить о мире в терминах объектов. Объектно-ориентированные методы разработки легли в основу идеологии сборки систем из отдельных компонентов; в качестве примера можно назвать такие технологии, как JavaBeans и CQM+.

Введение в язык UML

Унифицированный язык моделирования (UML) являетя стандартным инструментом для создания "чертежей" программного обеспечения. С помощью UML можно визуализировать, специфицировать, конструировать и документировать артефакты программных систем.

UML пригоден для моделирования любых систем: от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это очень выразительный язык, позволяющий рассмотреть систему со всех точек зрения, имеющих отношение к ее разработке и последующему развертыванию. Несмотря на обилие выразительных возможностей, этот язык прост для понимания и использования.

Изучение UML удобнее всего начать с его концептуальной модели, которая включает в себя три основных элемента: базовые строительные блоки, правила, определяющие, как эти блоки могут сочетаться между собой, и некоторые общие механизмы языка.

Несмотря на свои достоинства, UML - это всего лишь язык; он является одной из составляющих процесса разработки программного обеспечения, и не более того. Хотя UML не зависит от моделируемой реальности, лучше всего применять его, когда процесс моделирования основан на рассмотрении прецедентов использования, является итеративным и пошаговым, а сама система имеет четко выраженную архитектуру.

Язык UML предназначен прежде всего для разработки программных систем. Его использование особенно эффективно в следующих областях:

- информационные системы масштаба предприятия;
- банковские и финансовые услуги;
- телекоммуникации;
- транспорт;
- оборонная промышленность, авиация и космонавтика;
- розничная торговля;
- медицинская электроника;
- наука;
- распределенные Web-системы.

Сфера применения UML не ограничивается моделированием программного обеспечения. Его выразительность позволяет моделировать, скажем, документооборот в юридических системах, структуру и функционирование системы обслуживания пациентов в больницах, осуществлять проектирование аппаратных средств.

Терминология и нотация

Вопрос терминологии в программной инженерии, а тем более РУССКОЙ 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.

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:

1. фигуры,
2. линии,
3. значки,
4. 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.

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.

Виды диаграмм UML

Прежде чем перейти к обсуждению основного материала этой лекции, 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 в предыдущей лекции, 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.

Почему нужно несколько видов диаграмм

Для начала определимся с терминологией. В предисловии к этой лекции мы неоднократно использовали понятия системы, модели и диаграммы.

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

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

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.

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".

Диаграмма- это графическое *представление множества* элементов.

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.

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 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).

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. Итак, мы кратко рассмотрим такие виды диаграмм, как:

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

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.

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. 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.

Рис. 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*.

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*.

Рис. 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.

Как вы думаете, что означает эта диаграмма (рис. 2.3)?

Рис. 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 (рис. 4). Думаем, вы сами, без нашей помощи, легко догадаетесь, о чем там идет речь.

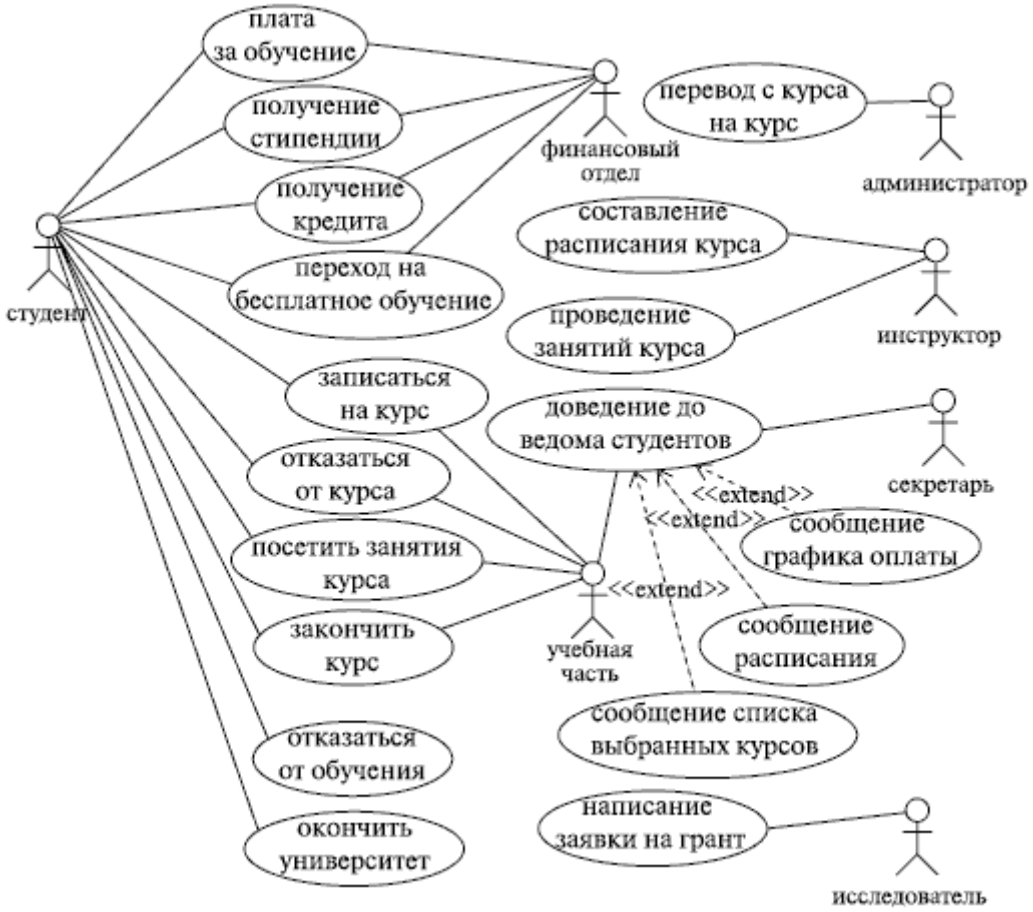

Рис. 4.

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.

Первый пример (рис. 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.

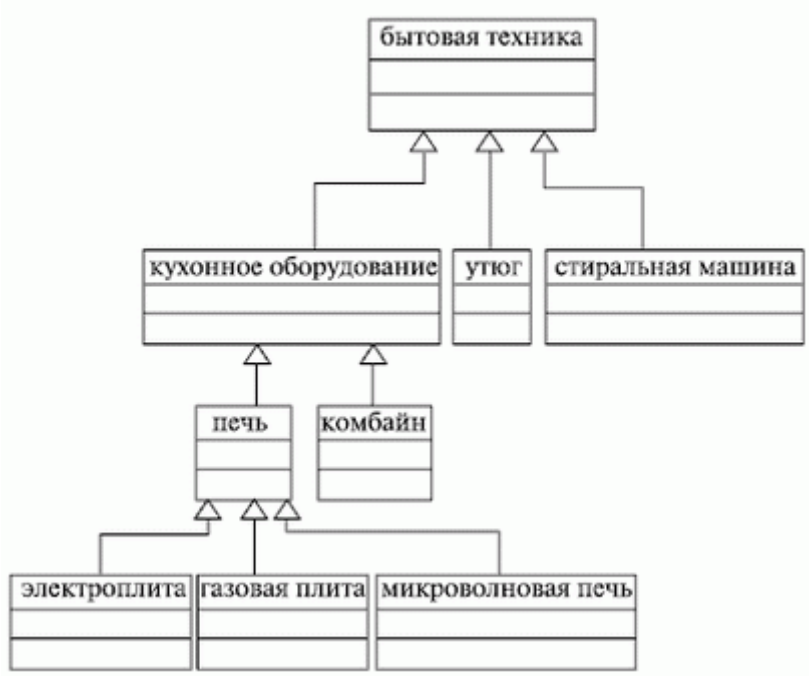

Рис. 5.

Рассмотрим еще пример (рис. 6):

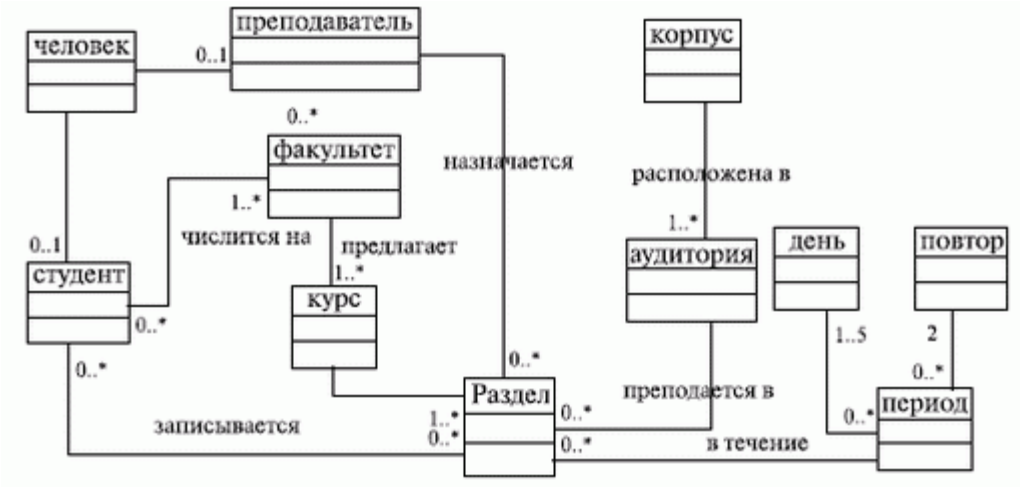

Рис. 6.

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 (рис. 7):

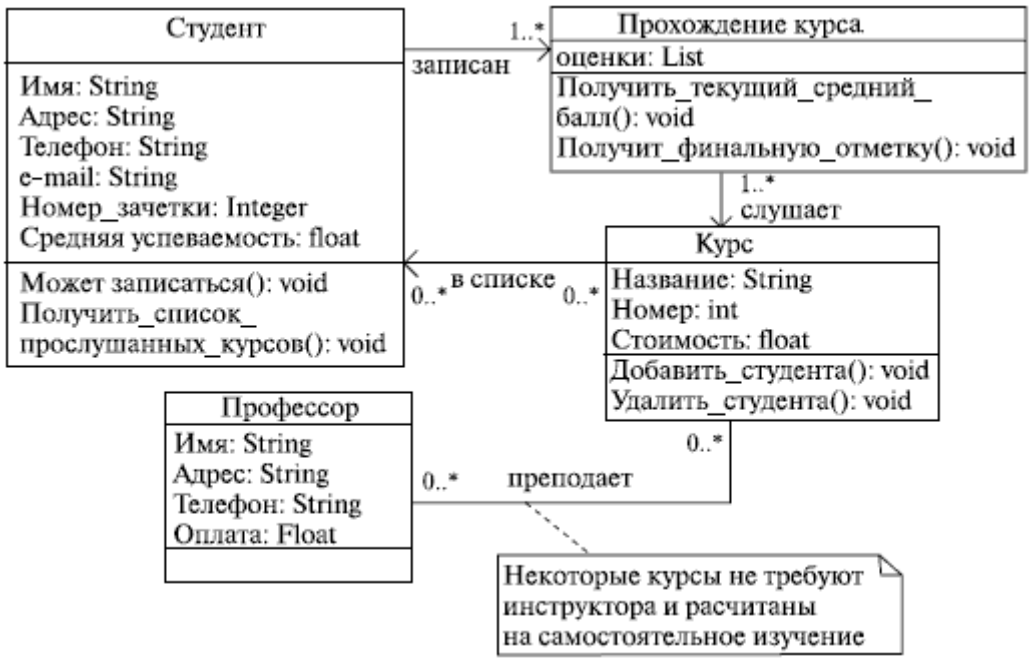

Рис. 7.

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.

Диаграмма объектов (object diagram)

А что такое объект?

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.

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 рис. 8.

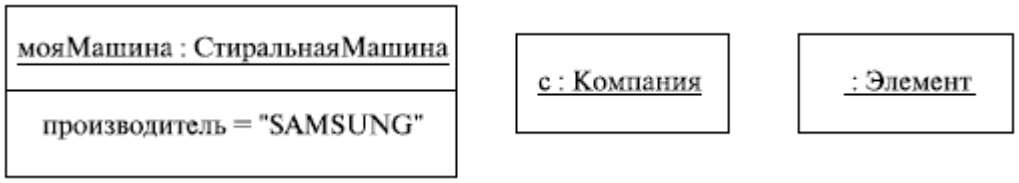

Рис. 8.

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 (рис..9).

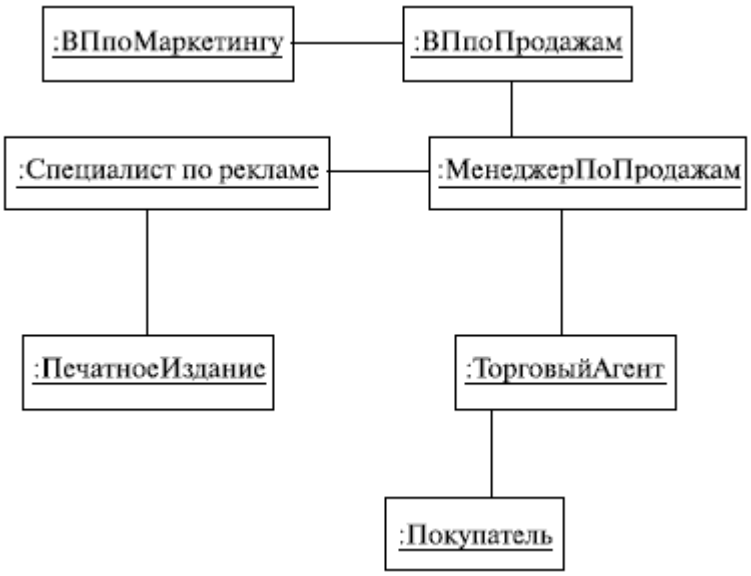

Рис. 9.

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.

Рис. 10.

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

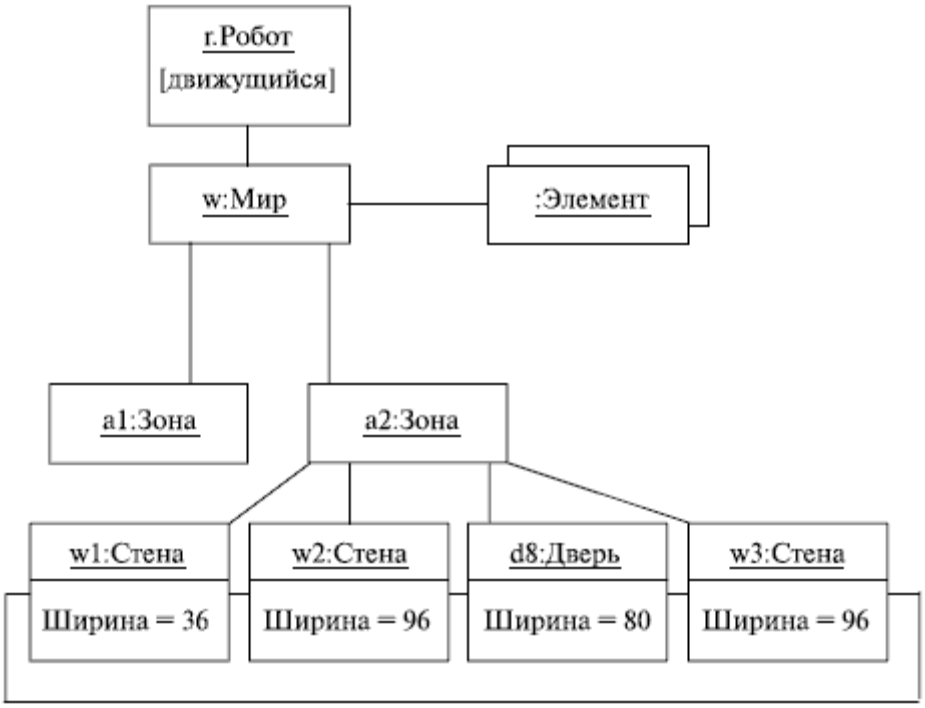

Рис. 11.

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.

Приведем пример диаграммы последовательностей (рис. 12):

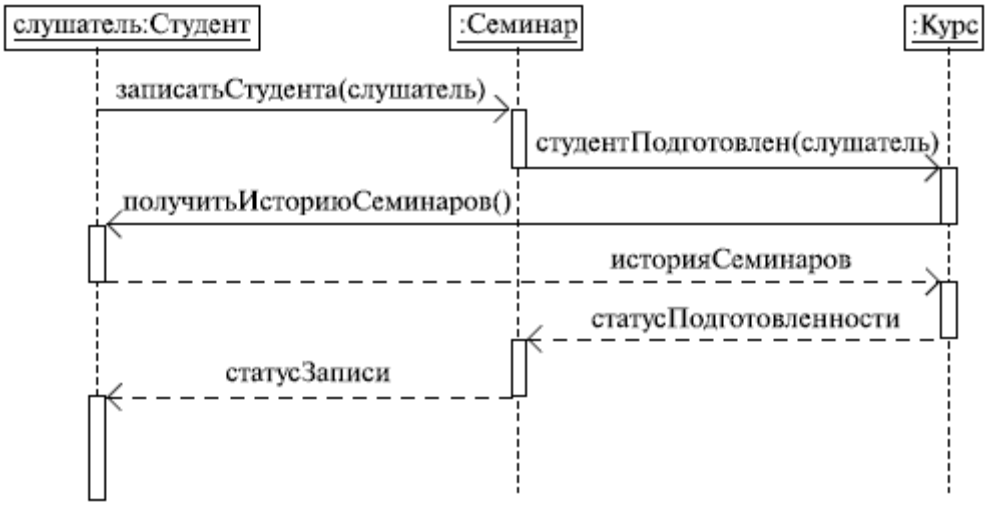

Рис. 12.

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 семинаров 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!

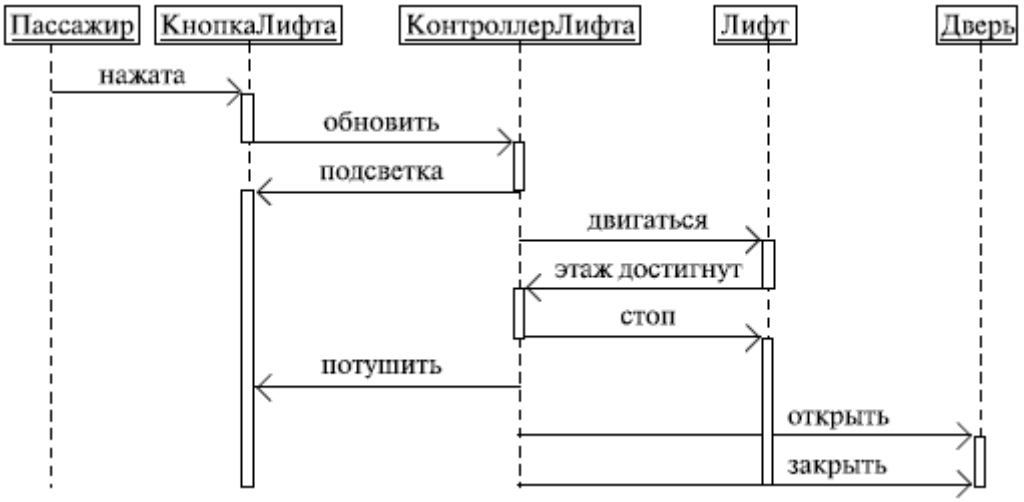

Рис. 13.

А вот что описывает следующая диаграмма (рис. 2.13), попробуйте догадаться самостоятельно. Только, чур, не подсматривать в нижеследующий текст лекции!

Ну как, догадались? А мы даже и не сомневались! Конечно, это же работа обычного домового лифта, которым мы пользуемся каждый день! Кстати, посмотрите на имена объектов - видно, что это уже несколько иной стиль проектирования, чем в предыдущем примере. И наконец, *еще один пример* (рис. 14):

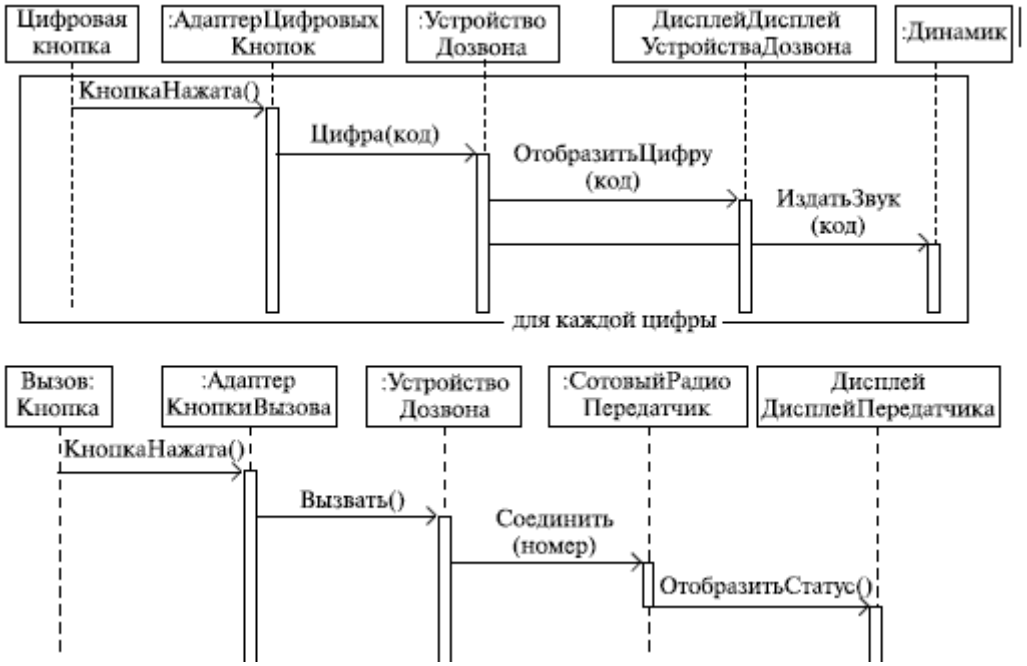

Рис. 2.14.

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 (рис. 15):

Рис. 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 (рис. 16):

Рис. 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.

И, наконец, еще один пример (рис. 2.17.

Рис. 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 (рис. 18):

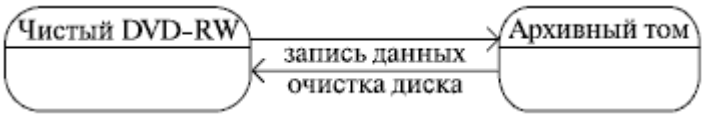

Рис. 18.

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

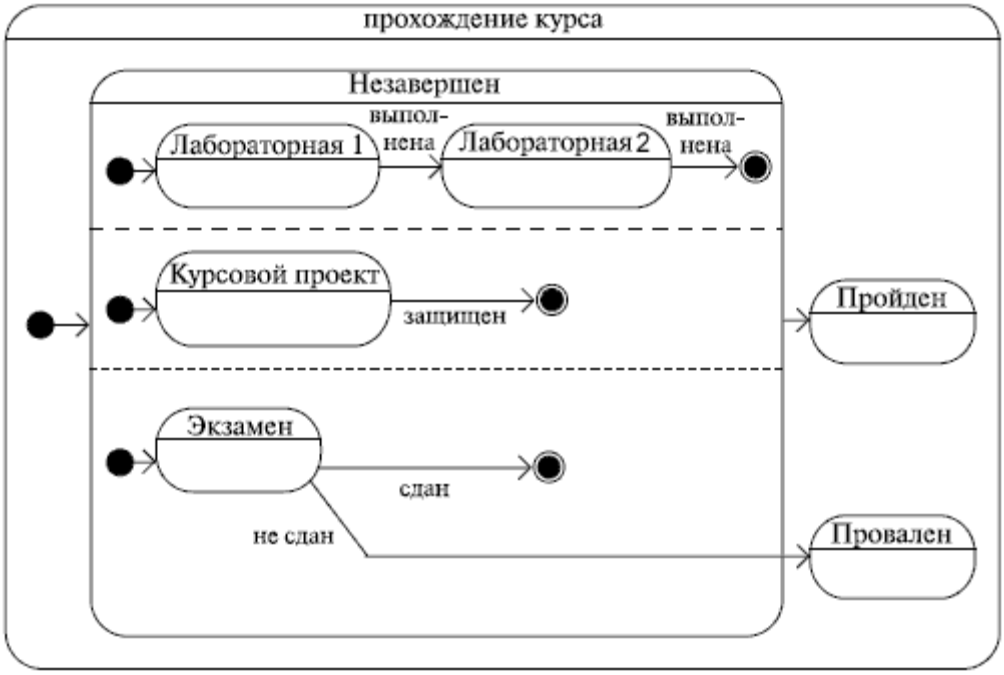

Рис. 19.

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 (рис. 2.20):

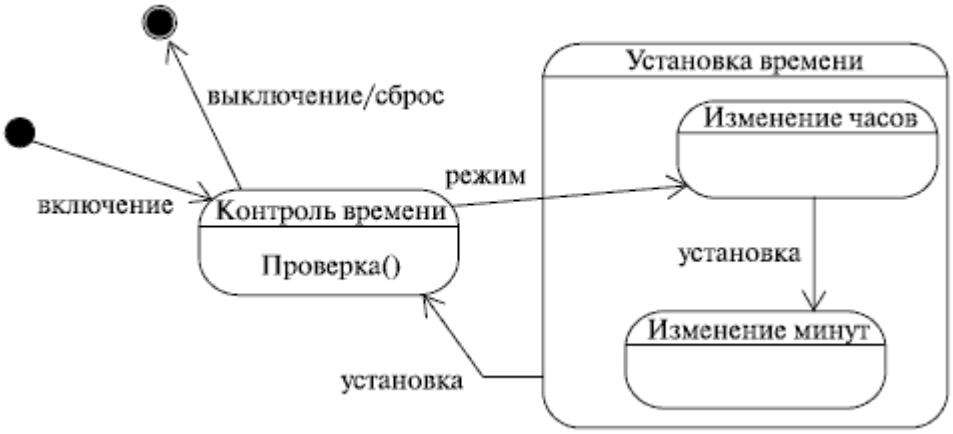

Рис. 20.

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 (рис. 2.21):

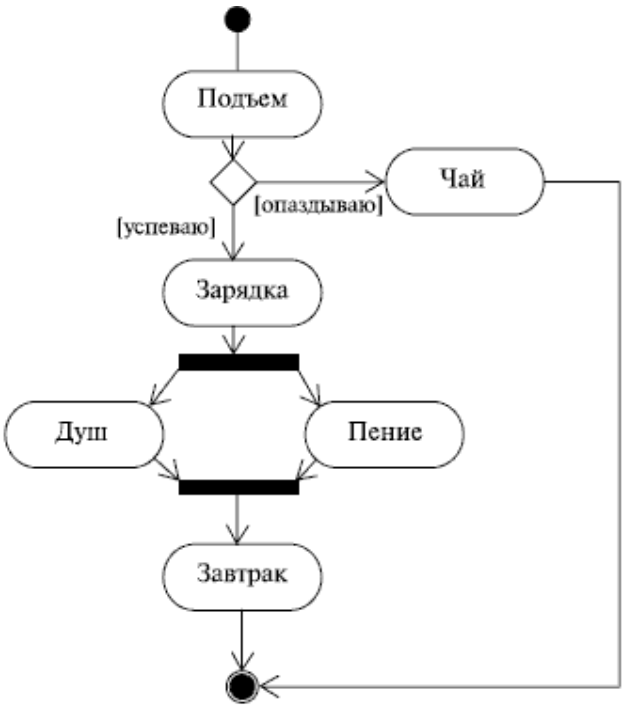

Рис. 21.

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! А вот еще пример(рис. 2.22):

Рис. 22.

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

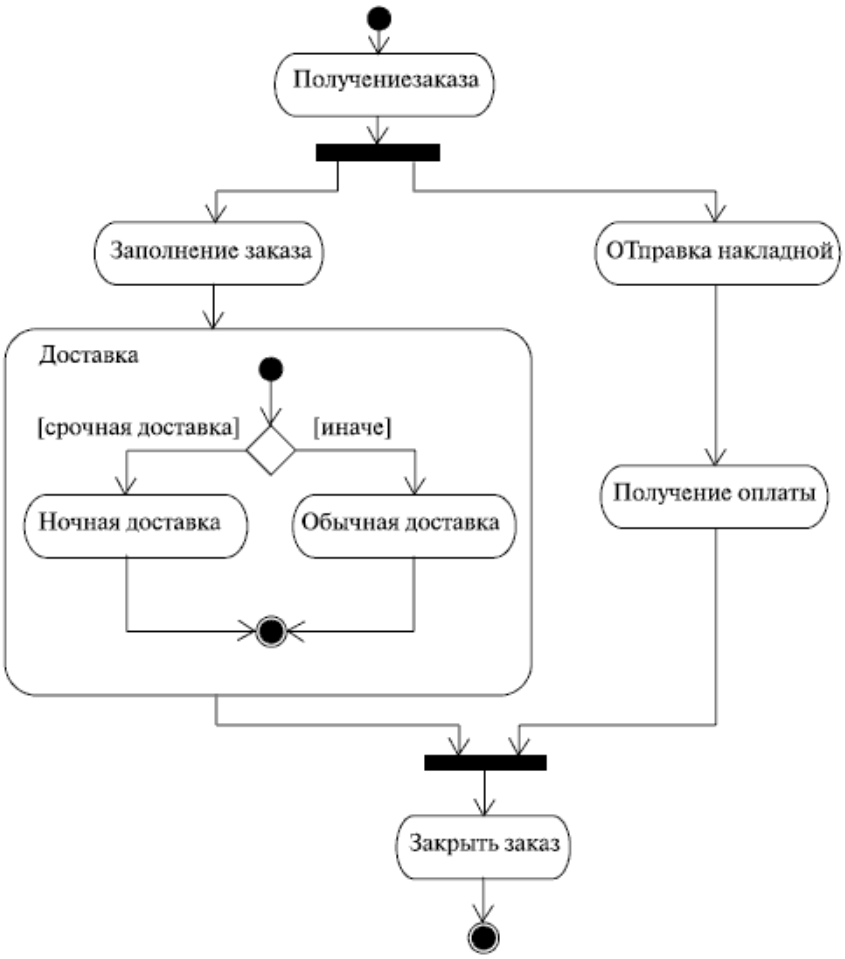

Рис. 23.

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 (рис. 24):

Рис. 24.

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

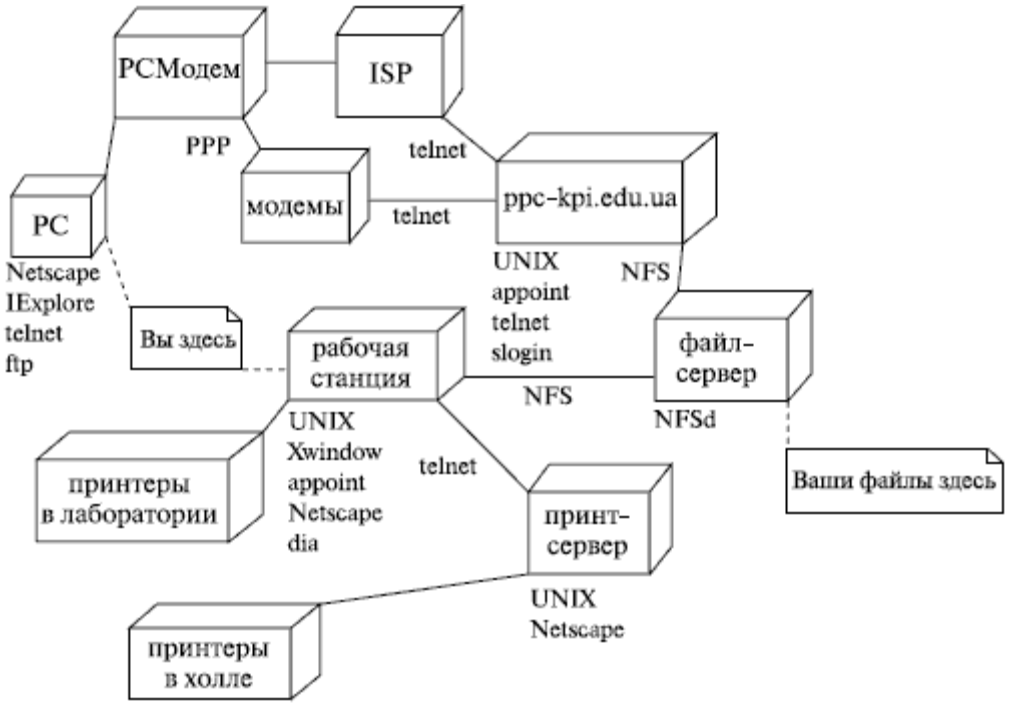

Рис. 25.

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

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:

- Какие именно виды диаграмм лучше всего отражают архитектуру системы и возможные технические риски, связанные с проектом?
- 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!

И напоследок еще 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 в одной из последующих лекций, пока же отметим, что все диаграммы, виденные вами в этой лекции, построены с помощью TAU G2 от Telelogic.

ЗАДАНИЕ

Разработать диаграммы прецедентов и активностей для предметной области проектируемой в соответствии с вашим заданием информационной системы.