2.2. UML – прожиточный минимум для программиста

По большому счету любой программист, не зная UML, знаком со многими компонентами UML на практике, поскольку реальности объектно-ориентированного программирования отражены в соответствующих компонентах. Кроме того, практика проектирования программного обеспечения в среде ООП формирует соответствующие абстракции. Рассмотрим подробнее.

Технология ООП работает на логическом уровне – абстрактных представлений и физическом – описание принципов работы и механизма реализации в языке. Например, на уровне реализации в Си++ класс – это набор скомпилированных функций-методов, объекты – области статической (именованной) или динамической (адресуемой указателем) памяти, в которой код интерпретирует объявленные данные объекта. На уровне последовательности команд вообще никакого объектно-ориентированного и структурированного кода вообще нет, ибо любой компилятор реализует компоненты ООП в рамках системы команд традиционной фон-неймановской архитектуры, т.е. в сущности, на уровне блок-схем.

На логическом уровне в процессе проектирования программист оперирует более абстрактными понятиями. Например, класс для него – описание некоторой сущности в виде набора данных и методов, реализующих элементы его поведения.

Поэтому при описании компонент UML для программиста есть постоянная возможность апеллировать к его знаниям принципов ООП. Например, отношение обобщения соответствует механизму наследования, а отношение реализации – созданию объекта класса, если класс рассматривать как порождающую сущность.

Компоненты UML

Модели UML покрывают все аспекты разработки системы, в том числе бизнес-процессы предметной области, функционал, представление данных, параллелизм и синхронизацию, архитектуру, логическую структуру ПО, физическую структуру системы и т.д.. Предметом моделирования является не только программная система, но и материальное окружение, а также сам процесс ЖЦ разработки.

Описание любой системы состоит из **структуры** (статики) и **поведения** (динамики). Структура состоит из элементов (сущностей) и связей между ними. На этой основе создан набор компонент UML:

- **сущности** - «имена существительные»**,** статические компоненты модели;
- **отношения** – взаимосвязи сущностей;
- **диаграммы** – описание некоторой части модели в виде набора компонент, диаграммы описывают структуру (статика) или поведение (динамика) системы;
- **общие механизмы** – семантика (смысл) за пределами определенной выше формальной модели, расширение UML:
  - **спецификации** (дополнения) – «задний план» (background), текстовое или формализованное описание деталей семантики, свойственной данному компоненту, вплоть до его реализации на формальном языке;
  - **стереотипы** - общепринятый подвид компонента UML;
  - **помеченные значения** – заданная пара «стандартное свойство - значение» для компонента, например **location = <имя>** обозначает метку размещения компонента;

- **ограничения –** общепринятые свойства компонента, ограничивающие его поведение или возможности использования, например, **complete** для обобщения (абстрактного класса) обозначает невозможность создания в модели новых производных классов помимо имеющихся;
- **«абстрактная сущность – экземпляр реализации»**, наличие в модели описания абстрактных сущностей, порождающих в динамике множества реализаций, в терминологии UML – **классификатор – экземпляр.**

Сущности

Сущности – имена существительные, основные элементы диаграмм UML:

- структурные сущности: *классы (активные классы), интерфейсы, прецеденты и кооперации, компоненты, узлы*;
- поведенческие сущности: *взаимодействие и автомат*;
- группирующие сущности – *пакет*;
- комментирующие сущности - **аннотации.** Включают в себя **текст**, не привязанный к элементам модели, **заметки** (Note) с текстом, связанные с элементами модели (NoteLink), прямоугольники, полуовалы и овалы с текстом;

Классификатор

Основой понятия абстрагирования является пара категорий **абстракция – экземпляр реализации.** В процессе разработки программист постоянно сталкивается с этой парой:

- *код программы – исполнение*: программа как код – экземпляр исполнения с конкретными входными данными, исполняемый файл программы – процесс в операционной системе;
- *тип данных – переменная:* тип данных как описание формы представления данных – переменная как область памяти, содержащая значение определяющего ее типа;
- *класс – объект как экземпляр класса*. При этом в представлении объекта как экземпляра класса есть некоторое лукавство. Память данных объекта действительно является экземпляром для типов данных его свойств, а программный код методов разделяется.

Для программиста просто необходимо распространить это понятие на другие компоненты описания системы (сигнал, прецедент, компонент, узел). В UML категории классификатор-экземпляр используются для следующих сущностей:

- класс – объект;
- интерфейс – реализация (присоединение интерфейса к классу);
- тип данных – переменная;
- сигнал;
- прецедент;
- компонент;
- узел.

В диаграммах модели могут быть описаны взаимосвязи и поведение как классификатора (абстракции), так и отдельных его экземпляров. В качестве примера рассмотрим двусвязный список и его представление в виде диаграммы классов и диаграммы объектов (рис.2-2).

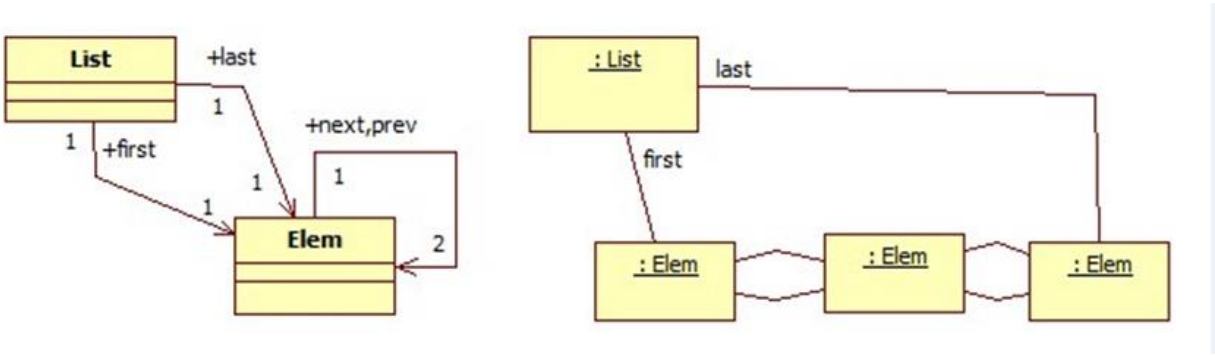

Рис.2-2. Диаграммы классов и объектов для двусвязного списка

Конкретная топология (линейность) в диаграмме классов **не может быть отражена,** т.к. она определяет взаимосвязи конкретных экземпляров класса Elem, например, двоичное дерево будет иметь аналогичную диаграмму классов.

Философия и программирование. Объективный идеализм (Платон, Гегель) рассматривает «абсолютную идею» как внематериальную сущность. Любой предмет материального мира является своего рода отпечатком этой идеи на косной материи. Вообще во многих философских, религиозных и художественных воззрениях идеальный *образ, замысел (промысел)* являются предтечей сотворения реальности. Ассоциации с вышеперечисленным налицо.

Стереотипы классов, активные классы

Стереотипы классов отражают реальности, с которыми сталкиваются программисты на уровне операционной системы или среды программирования:

- **process** – полноправный процесс в ОС (активный класс);
- **thread** – поток управления внутри процесса (активный класс);
- **exception** – исключение;
- **metaclass** – класс, объекты которого являются классами (описатель классов Class в метамодели Java);
- **utility**– класс с открытыми данными и методами (public);
- **powertype** – класс, генерирующий объекты любых своих производных классов (фабрика);
- **signal** – асинхронное внешнее событие;
- **type** – абстрактный класс, не создающий объектов, используется для спецификации структуры и поведения;
- **enumeration** – перечислимый тип.

В диаграммах устойчивости (Robustness) используются специальные значки для стереотипов классов – компонент программной архитектуры:

- **control** – ориентированный на управление (поведение, действие);
- **boundary** – ориентированный на внешнее взаимодействие;
- **entity** – ориентированный на данные.

Активные классы – классы, объекты которых являются самостоятельными потоками управления (Thread). Обратите внимание на одну тонкость. Активный класс, порождает поток (Connector и Client на рис.2-3), который может вызывать методы в других классах.

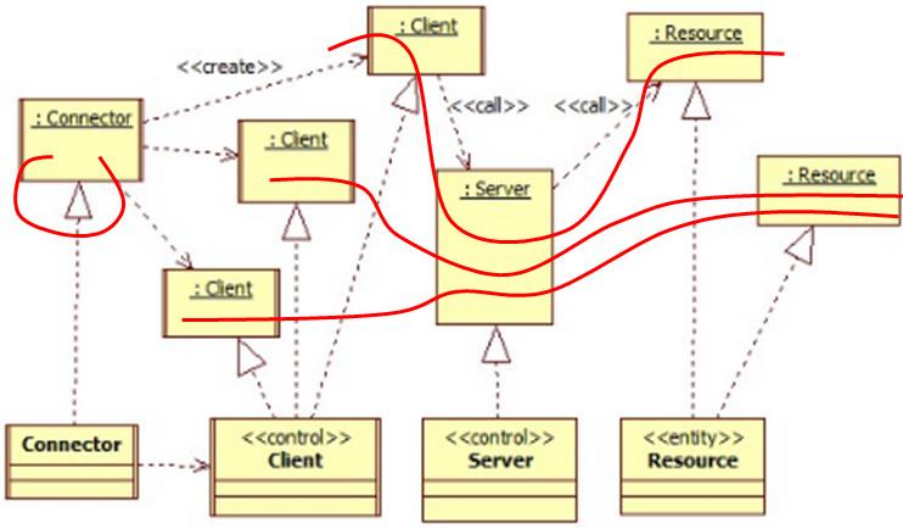

Рис.2-3. Активные классы

Несмотря на то, что метод будет вызываться «от имени» потока, содержащий его класс (Server, Resource) не будет являться активным.

Отношения

«ДОМ-2» — шоу отношений. ***Из интернета.***

Между описанными выше сущностями в системе устанавливаются связи – **отношения** (рис.2-4)**.** Связи могут быть различного типа: структурные, поведенческие, родовые (развитие и реализация). В любом отношении, даже если оно равноправно, выделяется ведущая и ведомая сущности (пара **источник – цель, причина - следствие**).

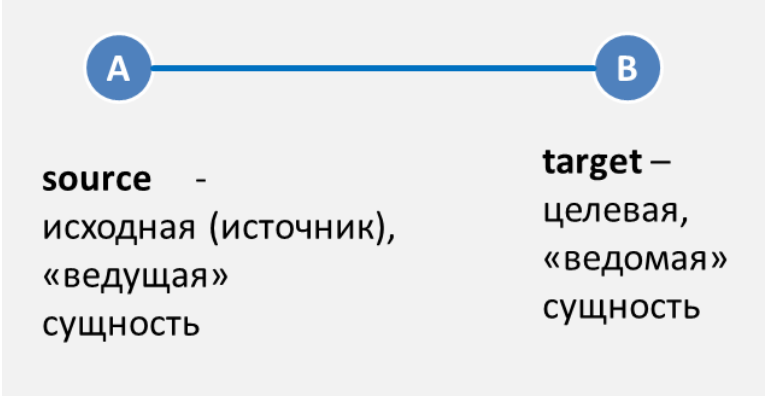

Рис.2-4. Сущность-источник и сущность-цель в отношении

Отношение вида **ассоциация** (рис.2-5) относится к структурной (статической) части описания системы. Ассоциация обозначает *постоянную существенную связь* между сущностями, имеющую место в течение всего времени существования системы.

Рис.2-5. Отношения: зависимость, ассоциация

Если ассоциация установлена между сущностями – классификаторами, то это означает установление ассоциации между отдельными парами экземпляров разных сущностей. В этом случае определяется **кратность** ассоциации, т.е. у экземпляра на каждом «конце» ассоциации оговаривается допустимое количество связей с *экземплярами противоположной сущности*: 0,1,* - любое, а также интервалы: 0..1,1..* и т.п..

Структурный характер отношения-ассоциации предполагает, что в любой момент времени возможен *выбор или переход по ассоциации от одного экземпляра к любому из экземпляров или к их группе в противоположной сущности.* Это является методологической основой ассоциации, отличающей ее от других видов.

Для программиста хорошей аналогией к отношению-ассоциации является взаимосвязь объектов в структуре данных – возможность одного объекта ссылаться на другой. Причем способ связи не имеет значения. Прямая ссылка, индекс, ключ в таблице с операциями линейного или двоичного поиска по ключу или хеширование – все это вписывается в UML в понятие ассоциации.

Тем не менее, в UML к ассоциации не следует подходить утилитарно, как средство высокоуровневого представления структуры оно не зависит от реализации. Ассоциация обладает следующими свойствами:

- указанная выше **кратность;**
- **имя** ассоциации в сущности;
- возможность **навигации (перехода)** по ассоциации в заданном направлении;
- **видимость** – модификаторы доступа private, public, protected, package, аналогичные принятым в любой среде ООП;

- возможность привязать к каждому экземпляру ассоциации объект **ассоциированного класса,** т.е. нагрузить ассоциацию некоторым функциональным объектом;
- определить **квалификатор –** ключевое поле перехода по ассоциации.

Хотя способ реализации ассоциации в UML принципиально не оговаривается, имеются виды ассоциаций – **агрегация и композиция,** которые указывают на «целостность» пары сущностей источника и цели:

- **композиция** - экземпляр источника «владеет» экземплярами целевой сущности, экземпляр целевой сущности создается/уничтожается источником и не может существовать вне связи с источником (символ композиции – закрашенный ромбик);
- **агрегация** – экземпляры источника и цели рассматриваются как единое целое по отношению к ассоциации, но принцип «владения» не применим, имеется возможность независимого существования экземпляров источника и цели (символ агрегации – прозрачный ромбик).

Наличие в системе объектов определенного класса не означает, что все они находятся в одинаковых структурных отношениях с другими классами. Например, на рис.2-6 имеет место фактически три различных вида объектов класса **B** в зависимости от того, к какой композиции они принадлежат.

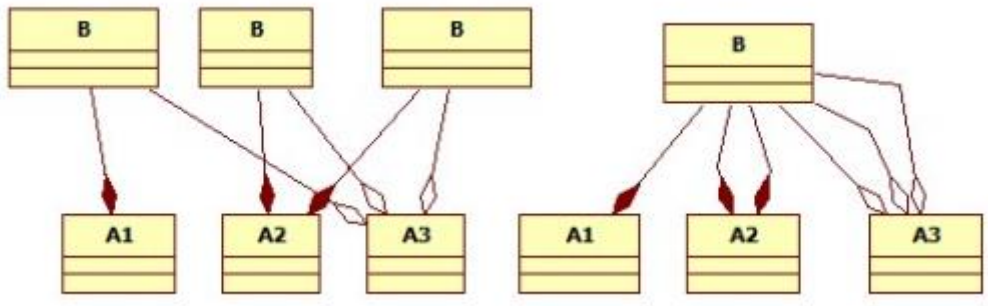

Рис.2-6. Агрегация и композиция: ограничения структурного представления.

Агрегации этих объектов с классом **A3** (левый рисунок) совпадают с соответствующими композициями в том смысле, что все объекты одной композиции могут принадлежать к одному типу агрегации. Если же объединить все группы объектов в один класс (правый рисунок), то такая взаимосвязь будет неочевидной. Можно сказать, что это свойство переходит на уровень **семантики ассоциации** и не отображается в структуре.

Отношение вида **зависимость** (рис.2-5) означает связь или взаимодействие сущностей, относящиеся к **поведению** системы, т.е. имеет место ограниченное по времени взаимодействие источника и цели. Также к зависимостям относятся разнообразные причинно-следственные связи, не относящиеся к структуре системы. Чтобы понять, что относится к зависимости, достаточно перечислить стереотипы – общепринятые типы зависимостей. Стереотипы зависимостей для классов:

- **bind** – экземпляр шаблона с параметрами;
- **derive** – данные источника являются фиктивными и вычисляется по данным цели;
- **friend** – исключение прав доступа, источник имеет исключительные права доступа к цели;

- **instanceof** – принадлежность цели (экземпляра) источнику (классификатору). В терминологии программирования это звучит так: цель является объектом класса-источника или одного из его производных классов;
- **instantiate** – источник создает экземпляр в цели. Зависимость установлена между классами: объект класса-источника создает объект класса-цели;
- **powertype** – источник генерирует объекты любых производных классов цели (см. 3.3, фабрика классов);
- **refine** – цель – детализация источника при проектировании, используется при документировании процесса разработки и версий проекта;
- **use** – источник использует семантику цели, например, класс-источник использует данные класса-цели;
- **call** – программный код источника вызывает код цели, вызывает метод;
- **trace** – источник является историческим потомком цели, аналогично refine, но при этом не происходит изменения источника, например, при переименовании.

Стереотипы зависимостей для объектов:

- **become** – цель – более поздний по времени объект-источник, позволяет отображать на диаграмме классов изменение структуры связей объектов во времени;
- **call** – код источника вызывает метод в цели;
- **copy** – клонирование, цель является копией источника.

Обобщение и реализация связаны с развитием сущностей или порождением их экземпляров (рис.2-7):

- **реализация** – целевая сущность представляет собой экземпляр исходной сущности – классификатора;
- **обобщение** - исходная сущность представляет собой производный класс (развитие) целевой сущности (базового класса).

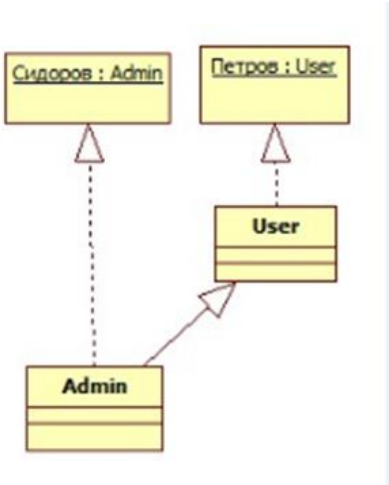

Рис.2-7. Отношения: обобщение и реализация

Диаграммы

Сущности и отношения являются компонентами, из которых строится описание системы. Единицей описания является диаграмма. Имеется значительное количество видов диаграмм, описывающих различные аспекты структуры и поведения системы. В то же время один и тот же вид диаграмм может использоваться в различных моделях на

разных этапах процесса разработки. Основой классификации является деление диаграмм на диаграммы описания **структуры** и **поведения** (рис.2-8).

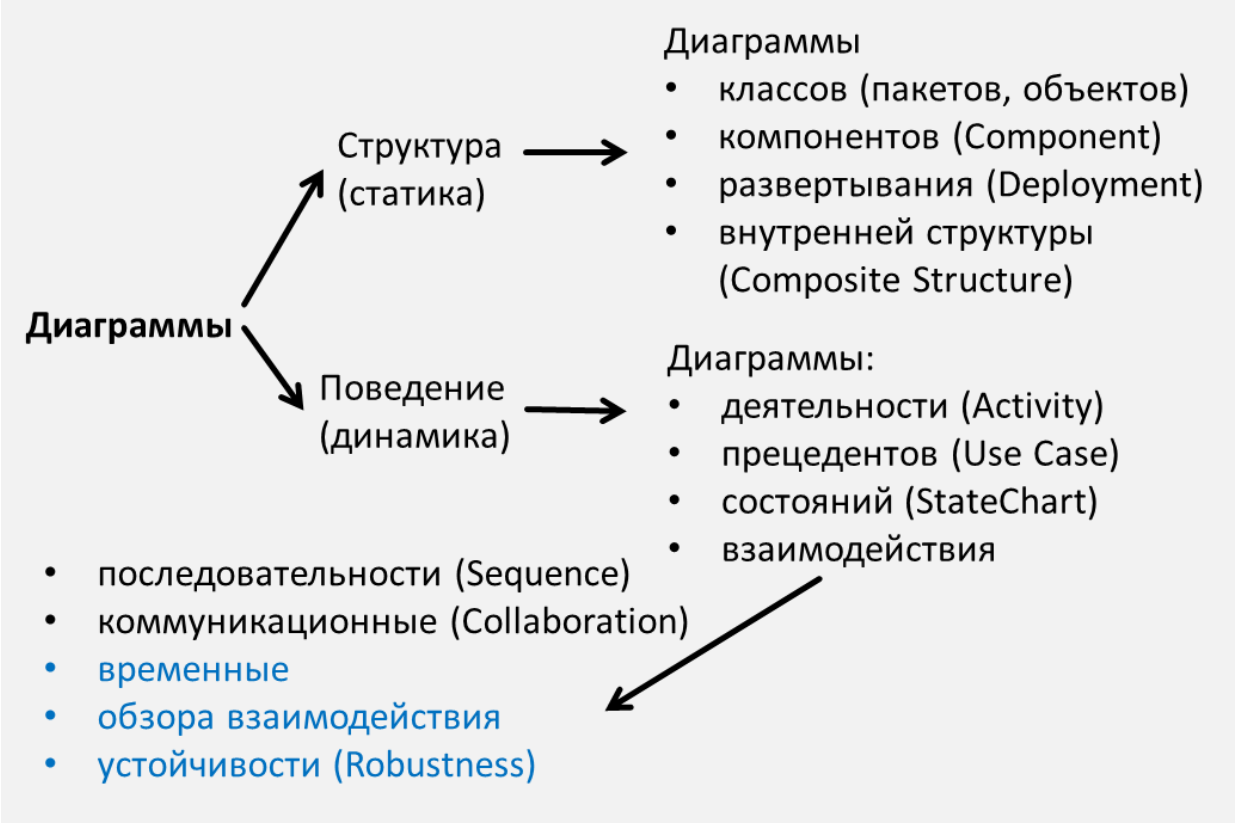

Рис.2-8. Виды UML-диаграмм

Диаграммы классов (Class)

Диаграмма классов наиболее понятна программисту, поскольку содержит значительное количество компонент, имеющих аналоги в ООП. Однако ее не следует понимать буквально, как исключительно описание структуры кода. Объектно-ориентированное представление является общепринятым для различных моделей в разных дисциплинах жизненного цикла. Среди них имеет место преемственность: модель – диаграмма классов на следующем этапе «наследуется» от предыдущего этапа. Виды моделей, использующие диаграмму классов:

- диаграмма классов предметной области (рис.2-9);
- диаграмма классов анализа;
- диаграмма классов проектирования;
- диаграмма классов реализации – структура программного кода (рис.2-10).

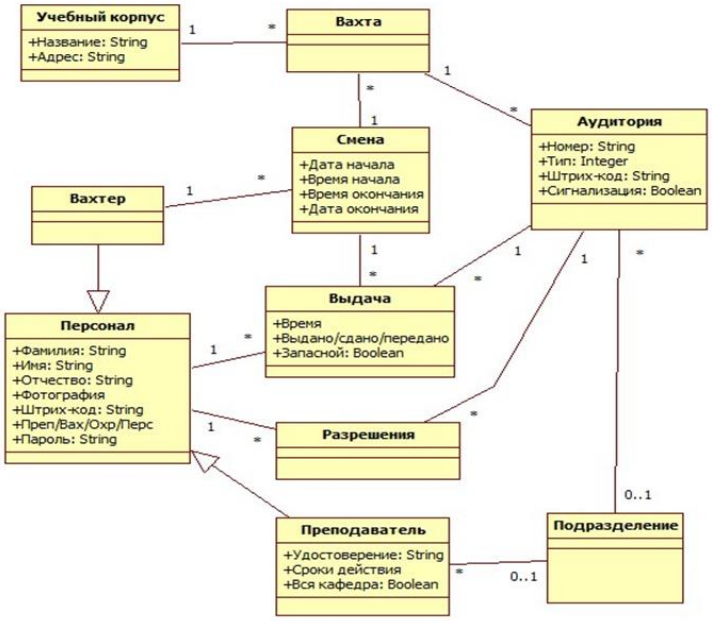

Рис.2-9. Диаграмма классов (модель предметной области)

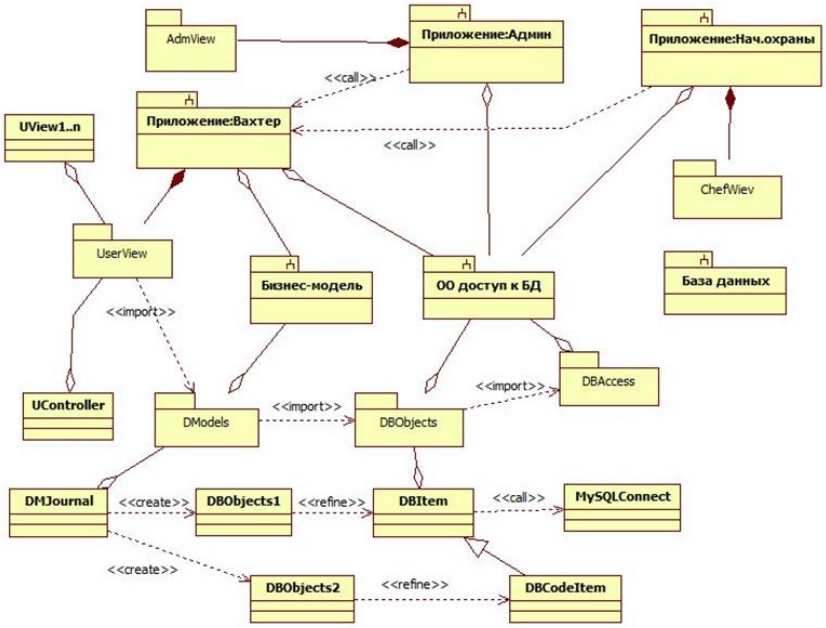

Рис.2-10. Диаграмма классов (структура программного кода)

Элементами диаграммы классов являются:

- классы, объекты, пакеты, подсистемы, стереотипы классов – исключения, сигналы;
- все виды отношений: ассоциации, агрегации, композиции, обобщения и реализации, ассоциированные классы;
- интерфейсы и классы-реализации, их присоединяющие.

Диаграммы внутренней структуры (Composite Structure)

Иногда удобно отойти от канонической объектно-ориентированной модели и рассматривать класс не как набор свойств-методов, а как набор взаимодействующих **частей,** а классы, в свою очередь, объединять в подсистемы по функциональному признаку. Тогда получается диаграмма внутренней структуры, элементами которой являются (рис.2-11):

- **подсистемы**;
- входящие в них классы;
- **части (part)** классов;
- **порты (port)** – именованные функциональные входы класса и подсистемы;
- **соединения (connector)** – связи между частями и портами;
- **интерфейсы и их реализации –** классы и части классов**;**
- **зависимости** между классами и их частями.

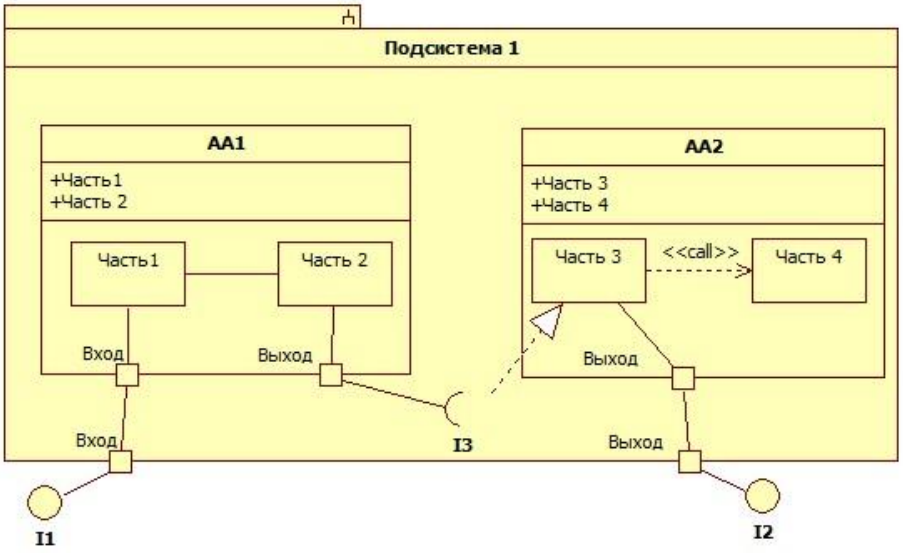

Рис.2-11. Диаграмма внутренней структуры

Диаграмма внутренней структуры отражает логическую структуру системы в виде модульной конструкции, состоящей из подсистем, классов, частей и их соединений. Фактически это общеизвестная **структурная схема,** привязанная по необходимости к классам.

Иногда элементы диаграммы внутренней структуры включают в диаграмму классов, тогда последняя интегрирует в себя диаграмму внутренней структуры.

Диаграммы компонентов (Component) и размещения (Deployment)

Диаграмма компонентов (рис.2-12) отображает структуру системы в виде составляющих ее программных компонент, компонент кода и используемых ими артефактов. Ее элементами являются:

- **компоненты** и **экземпляр компонентов** – физический модуль системы с содержанием, имеющим отношение к исполнению кода, имеет стереотипы **document, file, executable, library, table, source**. Специфические и важные компоненты называются **артефактами;**
- пакет программного кода;
- интерфейс, порт (**port**), соединение (**connector**), часть (**part**);
- отношения: ассоциация, зависимость, реализация.

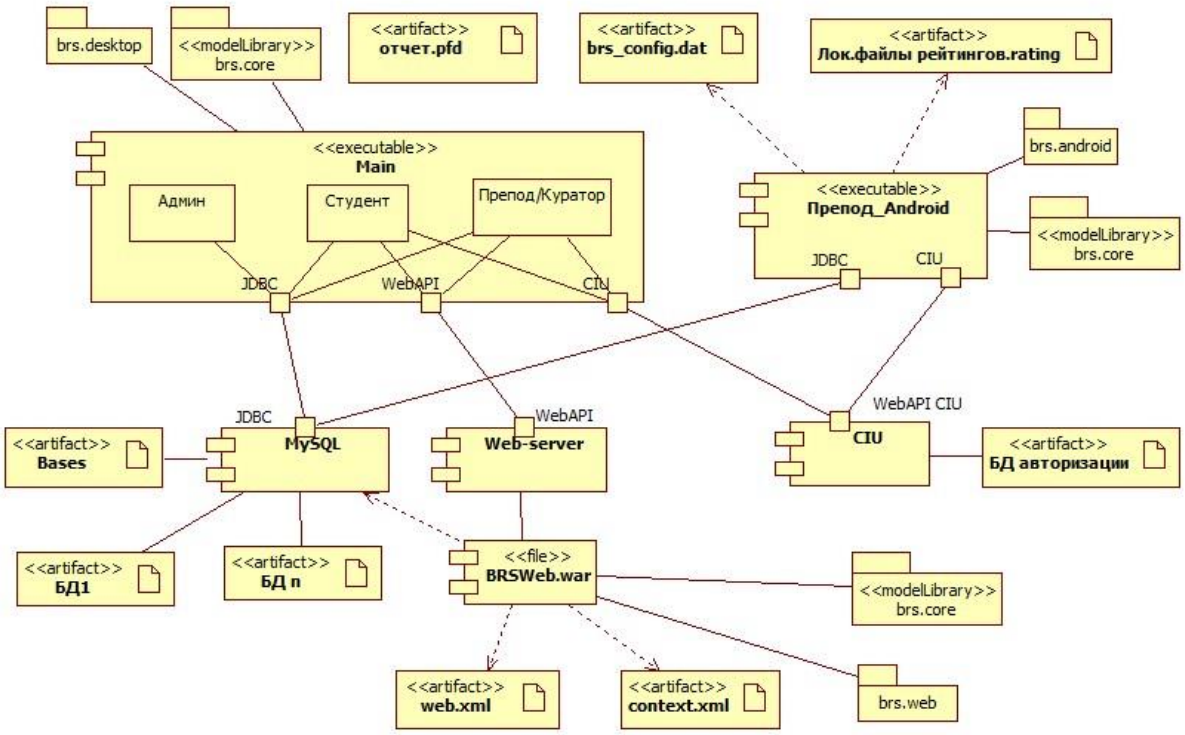

Рис.2-12. Диаграмма компонентов

Диаграмма размещения (рис.2-13) имеет тот же самый набор «выразительных средств», за исключением сущности - компонент: вместо нее фигурирует **узел (node)** – физическая единица программной системы (аппаратные и программные средства). Диаграмма изображает общую структуру размещения (узел – классификатор), либо конкретный вариант размещения (узел – экземпляр).

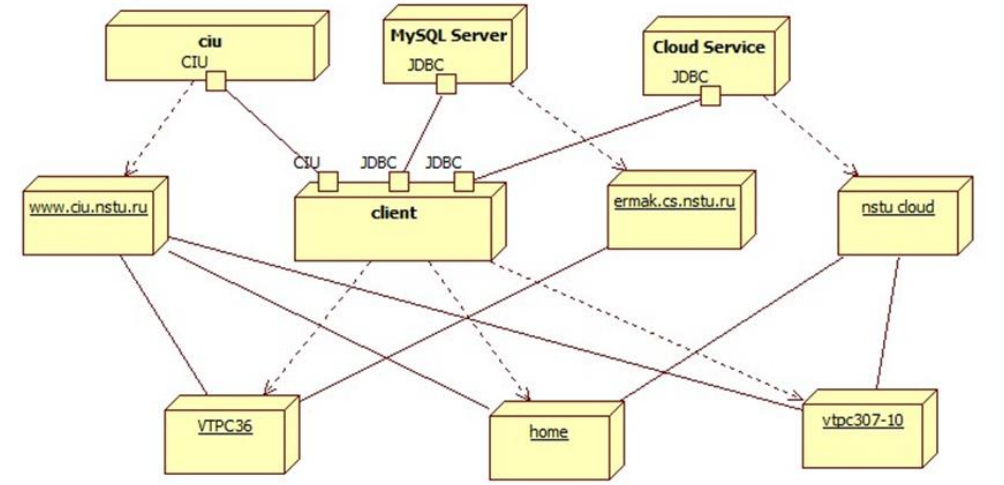

Рис.2-13. Диаграмма размещения

Диаграммы последовательностей (Sequence)

Основной иллюстрацией поведения системы является описание развертки ее работы во времени. Но фактор времени, как таковой, в моделях UML отсутствует, элементом описания поведения является **взаимодействие** как направленная причинно-следственная связь сущностей (или экземпляров).

На диаграмме последовательностей (рис.2-14) каждому экземпляру сущности (обычно объекту) соответствует «линия жизни», На линии жизни отображается

активность экземпляра, вызванная поступающими сообщениями. Сообщения могут интерпретироваться двояко:

- как передаваемые данные, в ожидании которых принимающий экземпляр не активен (ожидает, блокирован), т.е. в виде событийной модели;
- как переход потока управления из вызывающего экземпляра в вызываемый.

В соответствии с этим имеют место стереотипы сообщений:

- вызов, переход потока управления – **call**;
- возврат потока управления – **return**;
- передача сообщения – **send;**
- создание экземпляра – **create;**
- уничтожение экземпляра – **destroy.**

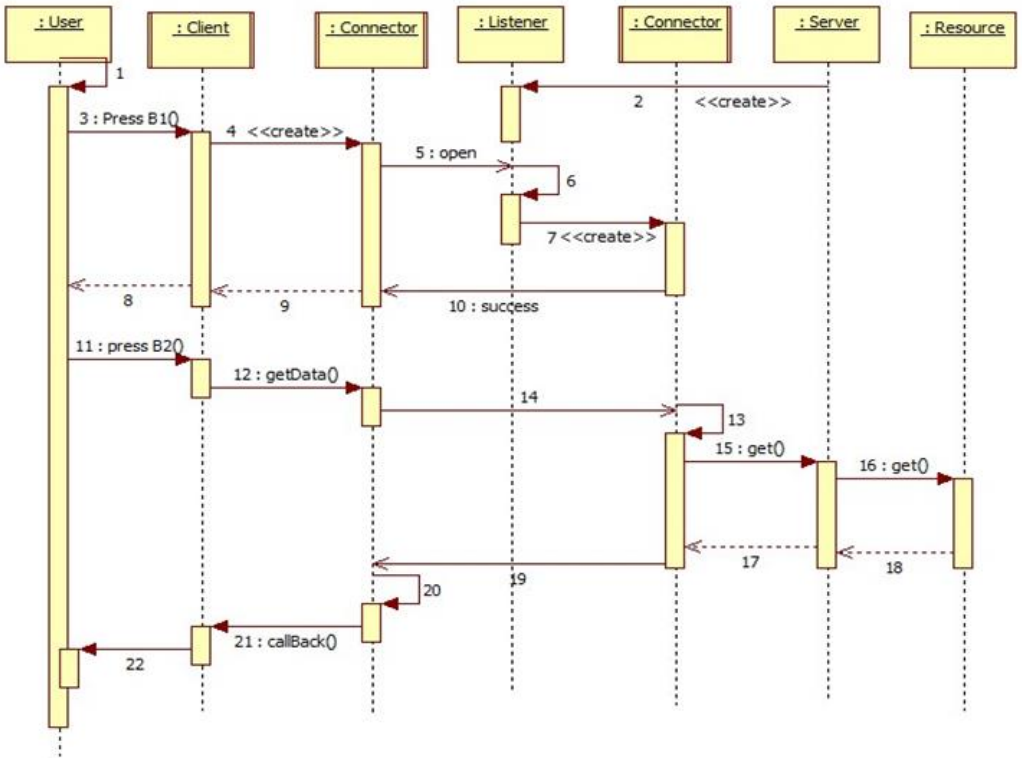

Рис.2-14. Диаграмма последовательностей

Диаграмма последовательностей может использовать сущности **роль классификатора (ClassifierRole)** со стереотипами **caseWorker, worker, internalWorker, boundary, control, entity, process** (рис.2-15). Такие диаграммы применяются для описания системы на уровне бизнес-процессов, функционала и архитектуры.

Часть диаграммы взаимодействия можно оформить в виде **оператора взаимодействия,** состоящего из частей – **операндов.** Это используется для отображения распространенных примитивов программирования. Перечислим некоторые из них:

- **alt -** ветвление**,** несколько альтернативных фрагментов, выбираемых по значению ключевого условия;
- **break –** досрочное завершение взаимодействия при выполнении условия;
- **critical/region -** критическая секция, синхронизируемый блок, аналогичный synchronized в Java;
- **ignore –** игнорирование любых сообщений, указанных в заголовках операндов;
- **consider –** обработка сообщений, указанных в заголовках операндов;
- **opt -** фрагмент, исполняемый при ключевом условии, ветвление без else;

- **par -** параллельный, все фрагменты выполняются параллельно;
- **loop –** цикл, фрагмент представляет собой тело цикла;
- **neg –** вызывается при обнаружении ошибки в процессе взаимодействия;
- **seq –** слабое следование, порядок исполнения операндов может быть любым, внутри операнда последовательность сохраняется;
- **strict –** строгая последовательность, все взаимодействия внутри каждого операнда для всех линий жизни не должны выходить за рамки операнда;
- **ref –** ссылка на другую диаграмму последовательности (включение).

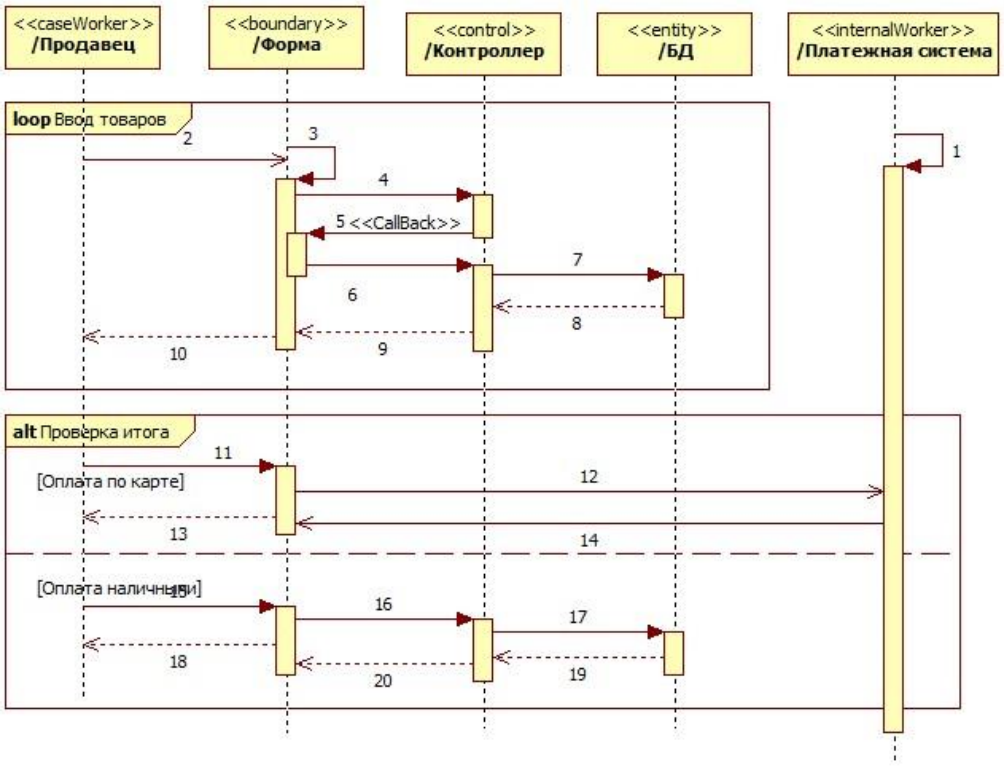

Рис.2-15. Диаграмма для ролей с операторами взаимодействия

Вся диаграмма может быть оформлена как комбинированный фрагмент **sd,** чтобы на него можно было ссылаться из других диаграмм.

Коммуникационные диаграммы (Collaboration)

Последовательность передачи сообщений между объектами (экземплярами) в ряде случаев можно отобразить непосредственно на диаграмме классов, если они передаются в соответствии с ассоциациями или зависимостями. На коммуникационных диаграммах (рис. 2-16) сообщения обозначаются нумерованными стрелками, привязанными к связям между экземплярами. Предполагается, что они являются элементами отношений, но характер их в данном случае не раскрывается.

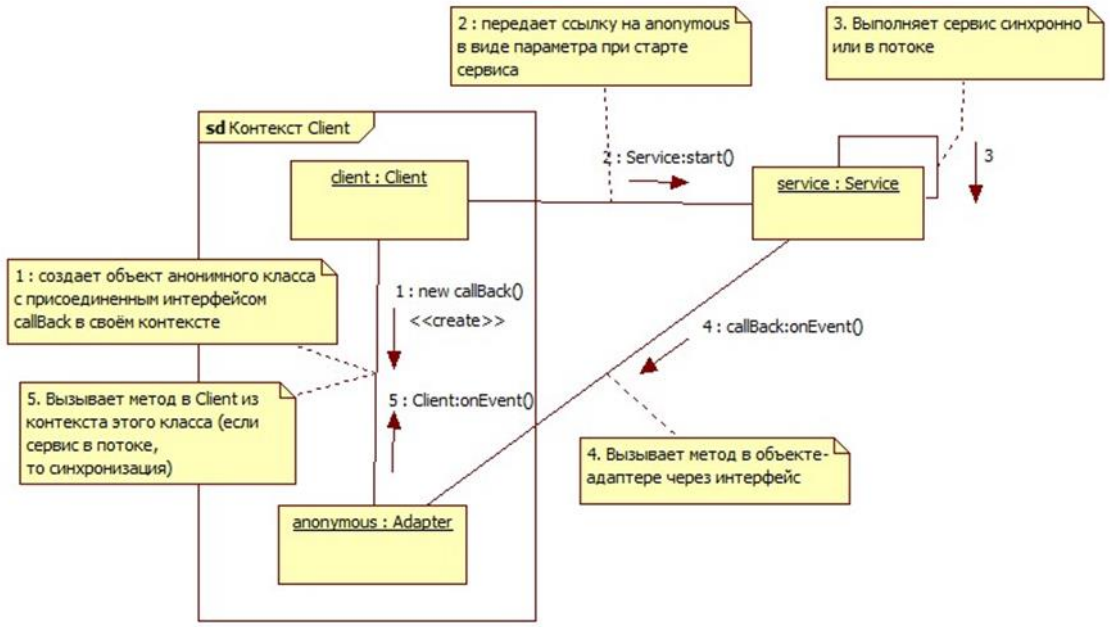

Рис.2-16. Коммуникационная диаграмма

Коммуникационная диаграмма может использоваться для экземпляров сущностей **роль классификатора (ClassifierRole)** аналогично диаграммам последовательности.

Диаграммы устойчивости (Robustness)

Диаграмма устойчивости (рис.2-17) является вариантом диаграммы классов, который применяется на этапах бизнес-анализа, функционального и архитектурного проектирования.

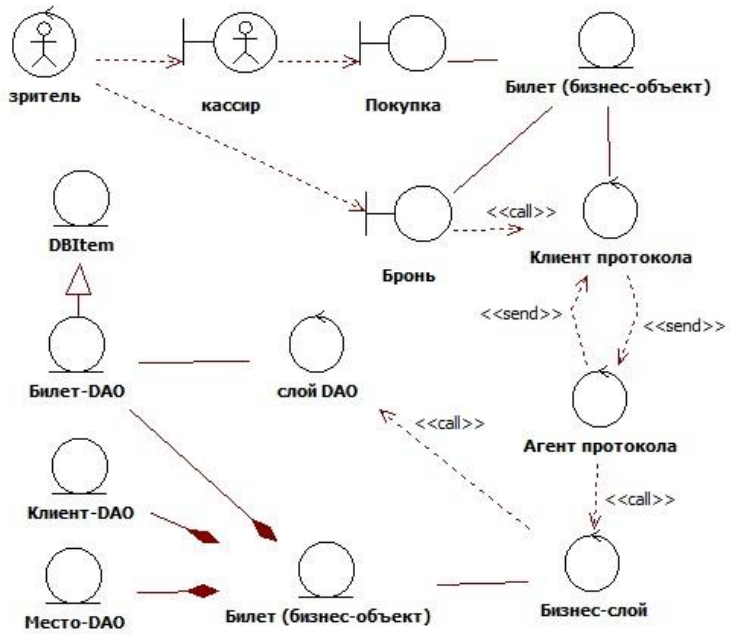

Рис.2-17. Диаграмма устойчивости

В качестве сущностей здесь фигурируют элементы внешнего окружения, а также элементы архитектурного описания системы, которые удается выделить на уровне бизнес- и функционального анализа. Это специфицируется стереотипами **ролей классификатора (ClassifierRole),** а на диаграмме соответствующими значками**:**

- **caseWorker –** сотрудник для связи с окружением, участник бизнес-процесса, непосредственно взаимодействующий с системой;
- **worker, internalWorker** – сотрудник, участник бизнес-процесса;
- **businnesEntity** – бизнес-сущность;
- **boundary-** – внутренний класс, ориентированный на взаимодействие с окружением системы;
- **control -** – внутренний класс, ориентированный на управление (действие, поведение);
- **entity** – внутренний класс, ориентированный на представление данных;

Это позволяет создавать модели с различным «процентным соотношением» описания функционала, архитектуры и реализации.

Диаграммы деятельности (Activity)

Диаграммы последовательности больше подходят для описания последовательных взаимодействий, когда активность в одном экземпляре является причиной аналогичной активности в другом. Если же требуется описать такие вещи, как равноправный параллелизм, взаимную синхронизацию параллельных ветвей и т.п., то наиболее адекватным средством является диаграмма деятельности.

Диаграмма деятельности представляет собой своеобразное сочетание формальной системы описания параллельного поведения автоматов – **сетей Петри** и традиционных блок-схем. Как известно, последние являются средством описания алгоритма в рамках отдельного потока управления.

Семантика диаграммы деятельности сложнее, чем у других диаграмм (рис.2-18). Постараемся ее описать в терминах, близких к ООП. Диаграмма состоит из узлов, соединенных дугами. Диаграмма одновременно работает в концепции **потока команд (управления)** и **потока данных (объектов).** Модель потока управления используется по умолчанию, модель потока данных – только в **объектных узлах** и смежных с ними узлах и дугах. При этом дуги, смежные с объектными узлами, обозначаются *пунктиром* (объектные дуги).

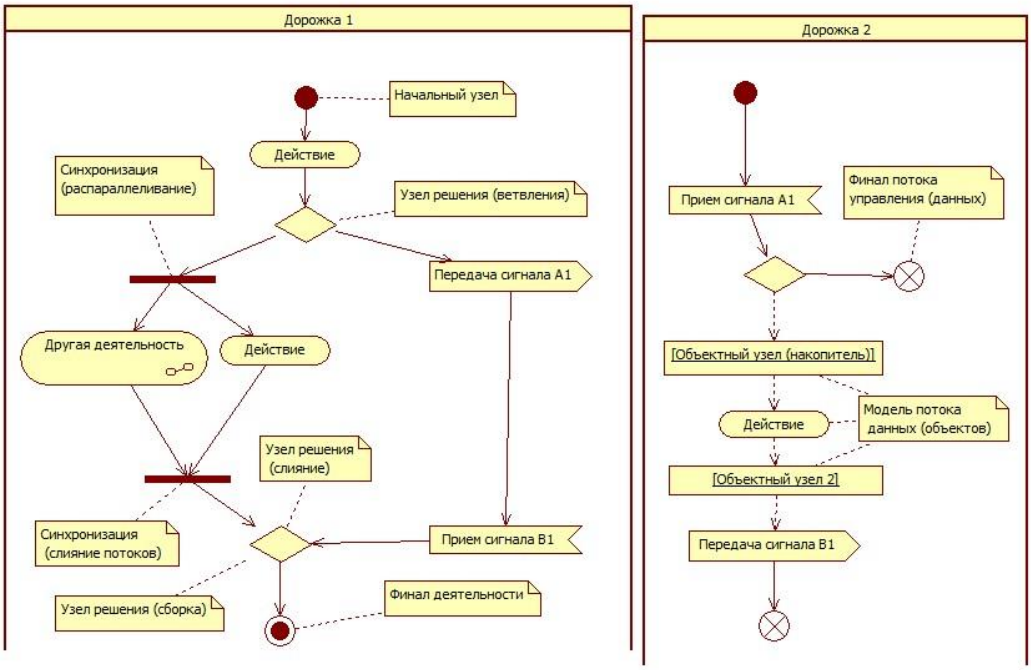

Рис.2-18. Основные элементы диаграммы деятельности

Рассмотрим подробнее. Модель в диаграмме деятельности является по определению многопоточной и использует понятие **маркера,** заимствованное из сетей Петри. Маркер (token) – это текущее состояние отдельного потока управления, привязанное к дуге диаграммы. В обычной блок-схеме последовательного алгоритма маркер – единственный, в диаграмме деятельности подразумевается, что количество их может быть любым в зависимости от интенсивности активизации деятельности, т.е. поступления маркеров в начальный узел.

Узлы действия являются *синхронизированными,* т.е. последовательно обрабатывают исполняющие их потоки, перемещая маркеры из входных дуг в выходные. Таким образом, маркеры могут накапливаться на входных дугах узлов, что соответствует синхронизации потоков к единичным (неделимым) ресурсам. Справедливости ради отметим, что сами маркеры на диаграмме не отображаются, т.е. диаграмма описывает не какое-то текущее состояние деятельности, а деятельность в целом.

Все вышесказанное, относится к моделям, где в каком-либо ее фрагменте моделируется внутренний параллелизм, связанный с разделением идентичными потоками общих ресурсов. Значительно чаще моделирование ограничивается внешним параллелизмом, когда разные участки диаграммы отображают принципиально разные потоки (функциональные виды деятельности, физические сущности).

В объектных узлах работает другое представление. Маркер представляет собой объект (экземпляр сущности) и объектный узел способен их накапливать. Свойства и особенности объектных узлов:

- множественность потоков управления в объектном узле превращается в множественность объектов данных;
- объектные узлы используются для обозначения компонент, где происходит явное накопление данных, создаваемых различными потоками (очередей, пулов);
- иногда объектные узлы не означают явного накопления, а используются для обозначения факта присутствия на данном этапе деятельности указанного типа объекта.

В диаграмме деятельности имеются следующие виды узлов потока управления:

- действие;
- вызов деятельности – исполнение указанной диаграммы деятельности как целого;
- начальные узлы деятельности, в каждом из них в начале цикла моделирования помещается по маркеру;
- финальный узел потока управления – завершает поток управления (уничтожает маркер), не завершая деятельности в целом;
- финальный узел деятельности – достижение его маркером завершает деятельность в целом;
- узел решения (решающий) – выбор одного из альтернативных направлений по выходным дугам, сборка маркеров по входным дугам, т.е. количество потоков при срабатывании узла *не меняется*;
- узел синхронизации – распараллеливает входной поток на несколько выходных (по выходным дугам), соединяет (синхронизирует) несколько потоков по входным дугам в один.
- узлы передачи и приема сигнала – предполагают передачу сигнала в некоторую внешнюю среду, а также ожидание его приема оттуда.

Замечание по теме. Узел действия при наличии нескольких входных или нескольких выходных дуг играет роль узла синхронизации по отношению к маркерам в этих дугах.

Еще одним элементом диаграммы является **дорожка.** Она соответствует физической сущности, роли, классу, в рамках которого осуществляется включенная в него деятельность.

В диаграммах деятельности, особенно если это касается моделирования бизнес-процессов и функционала, нельзя отождествлять модель с приведенными аналогиями из области программирования и реальностями реализации. Дуга, соответствующая потоку управления, легко может переходить с дорожки на дорожку, например, от дорожки заказчика к дорожке интернет-магазина. На функциональном уровне моделирования это вполне допустимо, поток управления в модели на разных шагах может иметь различную физическую природу.

В моделях бизнес-процессов и описания функционала диаграммы деятельности могут быть удобным средством спецификации сценариев (прецедентов) и моделировании предметной области. Что же касается моделей проектирования и конструирования, то иллюстрация шаблонов параллелизма с помощью диаграмм деятельности является не только тяжеловесной, но иногда и некорректной с точки зрения формального описания поведения модели в диаграмме. Например, при моделировании передачи сообщения с тайм-аутом (рис.2-19) в первом приближении все правильно, имеются два потока: обмена и тайм-аута.

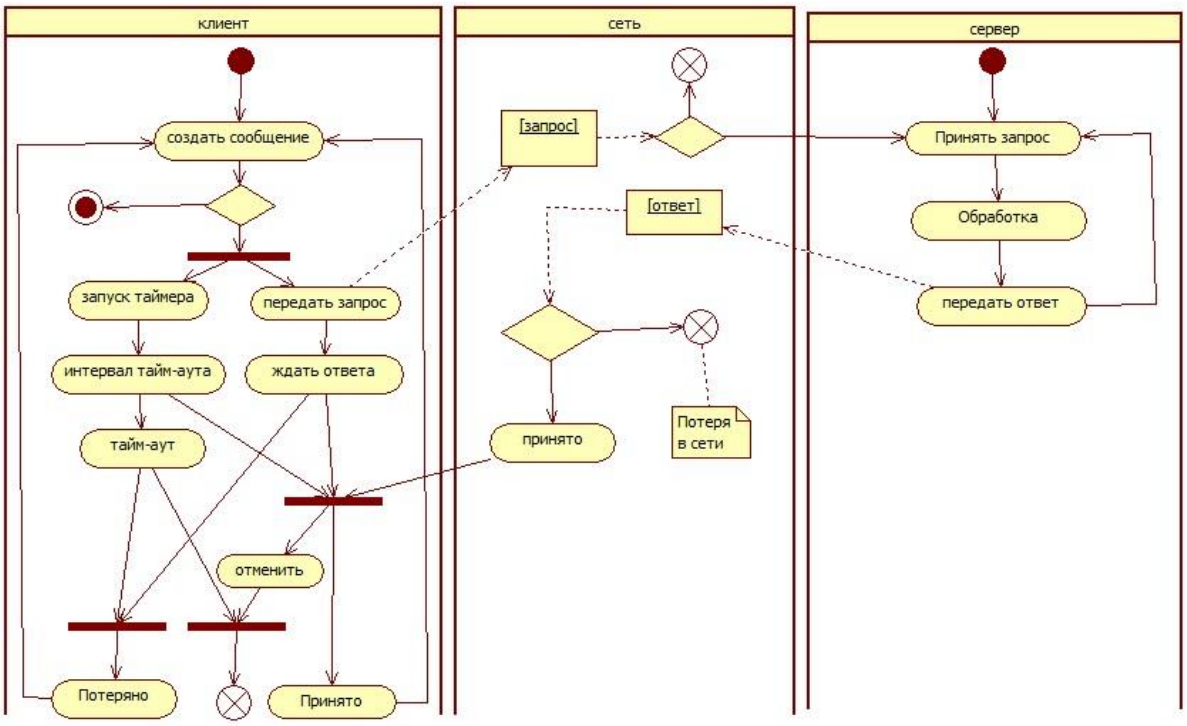

Рис.2-19. Диаграмма деятельности: передача сообщения с тайм-аутом

При приеме ответного сообщения тайм-аут отменяется. Однако это происходит в модели по истечении тайм-аута, *но не сразу*. Это связано с тем, что в данной модели не отражен процесс прерывания действия «интервал тайм-аута». Добавление соответствующих средств (область прерывания в UML2) еще более утяжелит ее.

Диаграммы состояний (Statechart)

В основе диаграммы состояний лежит модель конечного автомата (КА). Распространенной графической интерпретацией автомата является система состояний/переходов. Переход из состояния в состояние происходит по некоторому условию (символу) и сопровождается генерацией выходного сигнала (символа).

Фактически конечный автомат представляет собой программу, лишенную данных. Единственный элемент памяти автомата – его текущее состояние**.** Именно поэтому его диаграмма состояний более полно визуализирует его поведение. Неявная логика работы программы может определяться также текущим состоянием переменных, чего лишен автомат.

Конечные автоматы используются в программировании для описания поведения сущностей (классов), управляемых событиями от нескольких источников. Объект класса имеет набор состояний, относительно которых определяется его поведение и производится манипулирование им при обработке событий со стороны других сущностей.

Естественно, что КА в диаграмме состояний отличается от канонической модели в сторону большей технологичности. Основной набор элементов (рис.2-20) включает в себя:

- начальное и конечное состояние;
- простое промежуточное состояние с наборами действий при входе в состояние (EntryActions), выходе из состояния (ExitAction) и нахождении в состоянии (DoActions);
- переход с набором условий (Triggers) срабатывания и действий (Effects).

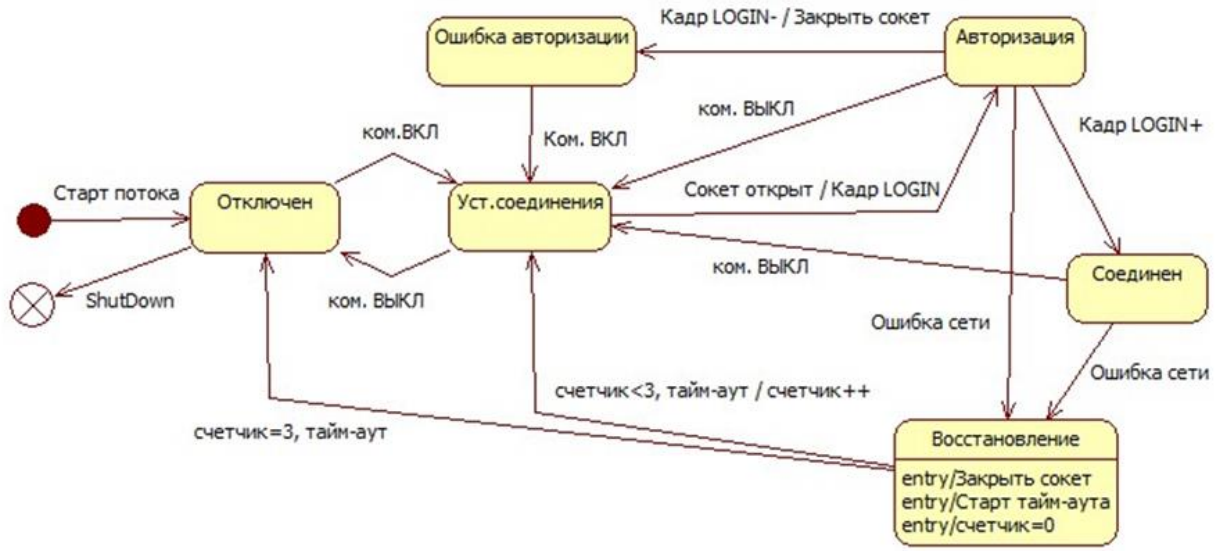

Рис.2-20. Диаграмма состояний

Дополнительные элементы диаграммы состояний связаны с введением в КА иерархии – составных состояний (рис.2-21). Составные состояния представляют собой на верхнем уровне одно состояние, которое раскрывается в виде отдельной диаграммы. При этом возможно создание групп параллельных состояний, т.е. текущее состояние подавтомата может быть двойным, тройным и т.д..

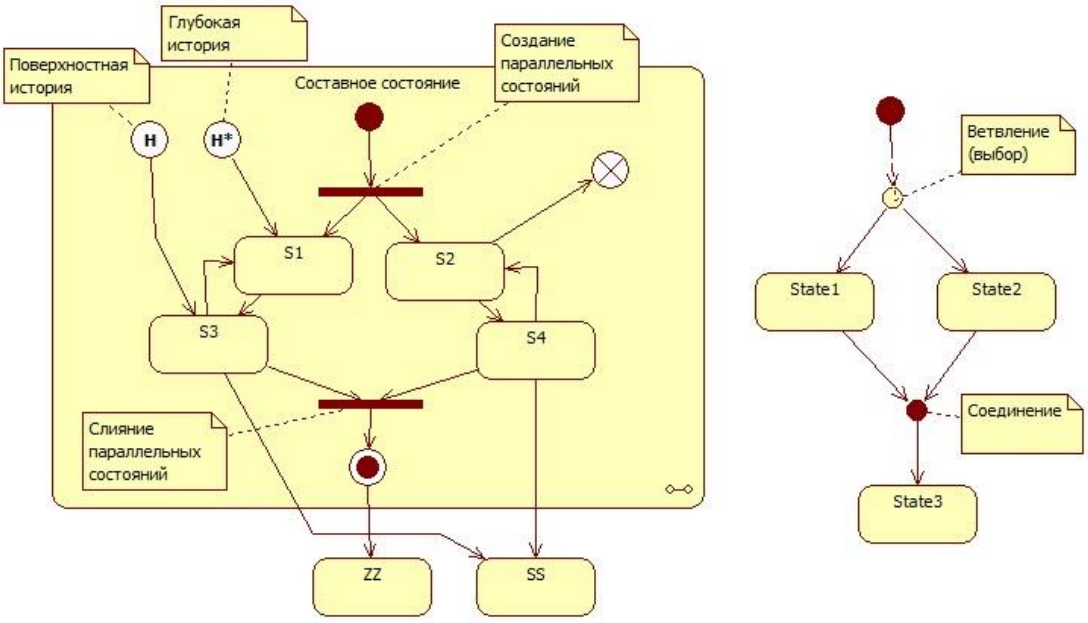

Рис.2-21. Дополнительные элементы диаграммы состояний

Выход из составного состояния может быть как через финальное состояние, так и помимо него. В последнем случае это выглядит как прерывание нормального поведения подавтомата. В этом случае при последующем входе автомат оказывается в состояниях **поверхностной** (ShallowHistory) или **глубокой истории** (DeepHistory).

Технологическим элементом являются точки ветвления (ChoicePoint) и соединения (JunctionPoint) состояний, которые позволяют «сэкономить» на одинаковых условиях и действиях в переходах.

Диаграммы прецедентов

Диаграмма прецедентов является фактически перечнем атомарных сценариев взаимодействия (прецедентов) и связанных с ними пользователей (актеров) (рис.2-22). Между овалами прецедентов и фигурками актеров устанавливаются связи – ассоциации. Между двумя прецедентами может быть установлена зависимость одного из видов:

- включение (include) – целевой прецедент является частью прецедента-источника;
- расширение (extend) – целевой прецедент выполняется при определенных условиях исполнения прецедента-источника;

Между двумя актерами или двумя прецедентами может быть определено отношение обобщения (наследования) – целевой прецедент является более общим, а источник – его расширением.

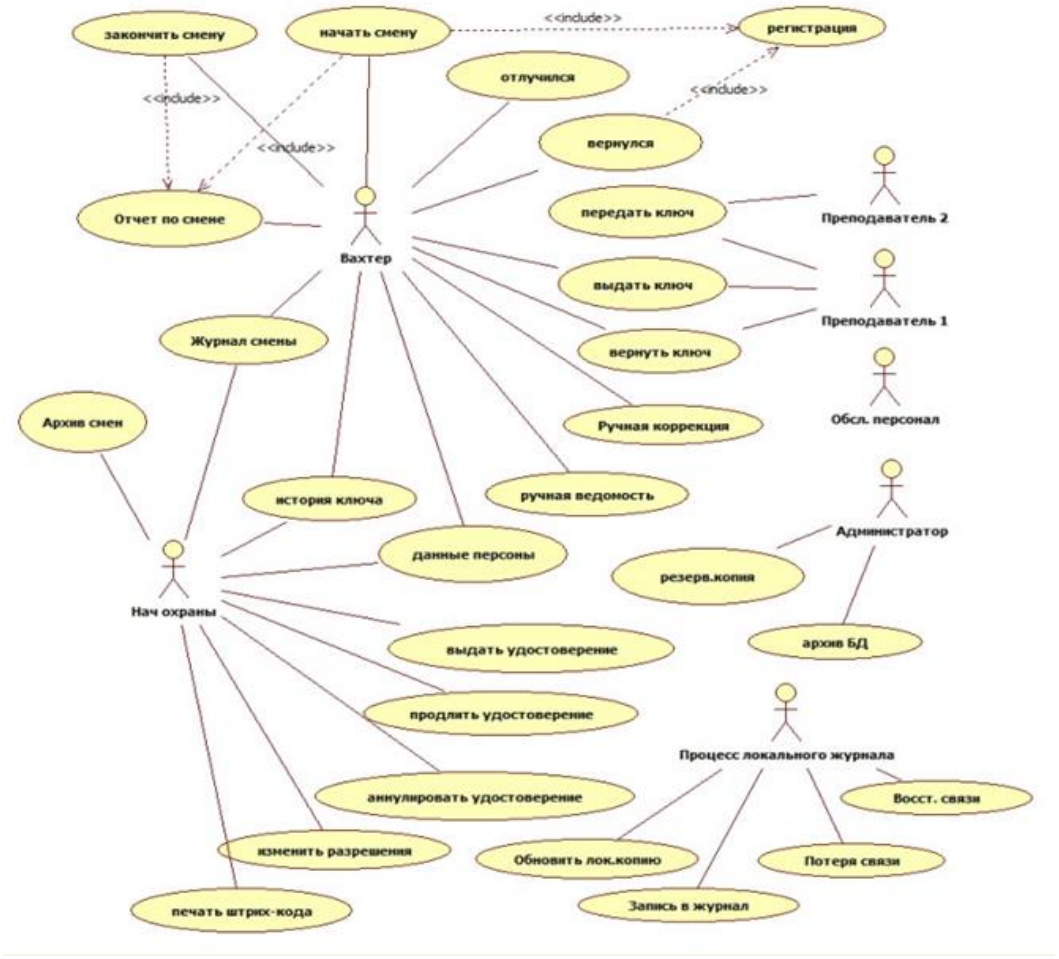

Рис.2-22. Диаграмма прецедентов

Замечание по теме. Актером может быть какая-либо внутренняя сущность или компонента системы, которая выступает как «виртуальный пользователь», инициирующий внятные сценарии. Например, периодически выполняемое архивирование данных может быть представлено актером-таймером, инициирующим соответствующий сценарий.